The Kronecker Booklet

Johann Blieberger

April 20, 2015

Chapter 1

Introduction

In this booklet we will mainly be concerned with verification of software properties. In particular, for single threaded software, we propose a method for checking whether an implementation complies to the specification of certain usage scenarios specified for modules the software consists of. We assume that usage scenarios are defined via finite state machines.

More accurately, we do not check usage scenarios against implementations but against control flow graphs of the implementations instead. Control flow graphs may be considered finite state machines for their own. Thus, finite state machines (FSMs) play a major rôle in our treatment.

FSMs can be represented by matrices. Verification of software, as presented herein, is primarily based on an operation defined for the matrices of the FSM's matrix representations. This operation is commonly known as *Kronecker product*.

Interestingly, another operation, *Kronecker sum*, itself defined via Kronecker product, can be used to model multi-threaded software. In addition, concurrent threads usually synchronize each other via *synchronization primitives*, like *semaphores* or *monitors*. Synchronization, then, can be modelled via Kronecker product. Properties of synchronized, multi-threaded software, like being free of *race conditions* or *deadlocks*, can be proved via Kronecker product based operations.

Summing up, we can verify several important properties of software systems via Kronecker product and related operators.

Since the analyzes presented herein are useful only if they can be done efficiently, we show how Kronecker algebra can be implemented such that it performs fast and without using too much memory. Thus we obtain time and space efficient operations via a lazy implementation of Kronecker operators.

Chapter 2

Finite State Machines

Modelling software systems can be done in various ways. One pattern shows up in different places with different names, but the essentials are always the same. These are the so-called finite state machines. Similar concepts can be found in finite automata, for flow graphs, and in many others.

2.1 Matrix Representation

Finite State Machines (FSMs) consist of a finite number of states. Some of these states serve a special purpose. For each FSM there is one *initial state*. For example in the graphical representation of an FSM in Figure 2.1, State 1 is the initial state, which is shown by a small arrow targeting State 1 and the text "start" being written to the left of the arrow. A certain number of states serve as *final states* of an FSM. In Figure 2.1 there is only one final state, namely State 2, which is shown by a double circle around the name of the state. All the other states and the initial state have single circles around their names.

Within this booklet we will always refer to states via numbers $1, 2, 3, \ldots$. More precisely, if an FSM consists of n states, these states will be numbered from 1 to n.

FSM states are connected via so-called *edges*. Edges have a direction shown in Figure 2.1 by arrows and they model *state transitions*. State transitions are used to illustrate how a system evolves over time. Usually, edges are labelled. Our example in Figure 2.1 has three edges:

1. edge $(1 \to 1)$ labelled with "a",
2. edge $(1 \to 2)$ labelled with "b", and
3. edge $(2 \to 2)$ again labelled with "a".

Edge labels may serve different purposes, depending on what our goal is. At the moment, it suffices to think that edge labels are "names" of the edges.

The system modelled by the FSM in Figure 2.1 may evolve as follows:

1. The system starts in State 1.
2. It evolves by a state transition $1 \to 1$, i.e., it stays in the same state, but it signals this transition to its environment by the edge label of edge $(1 \to 1)$, i.e., by producing the output "*a*".
3. The system transits to State 2 via edge $(1 \to 2)$ and produces the output "*b*".
4. It evolves by following edge $(2 \to 2)$ and produces output "*a*".

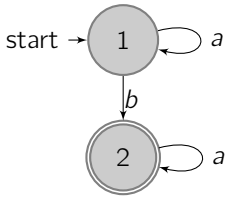


Figure 2.1: A Graphical Representation of a Simple Finite State Machine

5. The system terminates, which it is allowed to because the FSM is in a final state, namely State 2.

Representations of FSMs like that in Figure 2.1 will be used throughout this booklet. However, graphical representations are not the only way of representing FSMs. A different representation uses *matrices* and *vectors*. Let n be the number of states of a particular FSM. We may then choose a square matrix M of size n, i.e., a matrix having n lines and n columns. We fill the matrix with edge labels according to the following procedure:

1. *If the underlying FSM contains an edge* $(s \to t)$ *labelled by* e*, then we place* e *at the* s*th line and at the* t*th column of* M*.*

2. *If the FSM does not contain an edge* $(s \to t)$*, we set the place at the* s*th line and at the* t*th column to* 0*.*

For our example FSM the size of the matrix clearly is 2. Taking into account all edges and their labels, we obtain matrix M:

$$M = \begin{pmatrix} a & b \\ 0 & a \end{pmatrix}.$$

As we will have to study matrices with a very large number of zero entries, we will frequently write "$\cdot$" instead of "0" in order to facilitate focusing on the non-zero entries. Our matrix M thus reads

$$M = \begin{pmatrix} a & b \\ \cdot & a \end{pmatrix}.$$

Now we give some thoughts to the edge labels. In fact, we are going to define two operations for the edge labels.

The first operation, written in infix notation "$\cdot$", simply denotes juxtaposition of its operands. For example let "a" and "b" denote two edge labels, then "$a \cdot b$" denotes their juxtaposition. If it is clear from the context, what the edge labels are, we are free to write "ab" instead of "$a \cdot b$". We will do so frequently. In addition, we will use power notation to denote repetitive juxtaposition of identical operands. For example we will write "a^3" instead of "$a \cdot a \cdot a = aaa$" or "a^n" instead of n concatenated "a"s.

The second operation, written in infix notation "$+$", denotes an operation of choice. For example let "a" and "b" denote two edge labels, then "$a+b$" denotes that we are free to choose either "a" or "b". Programmers may consider the operands being the then- and the else-branch of an if-statement. Generalizing, we may also write "$\sum_{i=1}^{k} a_i$" if we want to describe a case statement consisting of k different cases. Or, we may write "$\sum_{x \in \mathcal{S}} x$" to denote a choice over a set $\mathcal{S}$.

It is, however, important to note that we assume $a+b = b+a$ for all labels a and b. This seems to be contra-intuitive for understanding $a + b$ as an if-statement, but seen from a higher perspective, the order of then- and else-branches of if statements is not crucial at all.

In addition, we introduce neutral elements for our new operations: We define $a \cdot 1 = 1 \cdot a = a$ and $a + 0 = 0 + a = a$ for all edge labels.[1] Furthermore, we assume that "+" is an idempotent operation, i.e., $a + a = a$ for all edge labels.[2] And, finally, we may write "$a^0 = 1$" and we note that "$a \cdot 0 = 0 \cdot a = 0$" for all labels a.

Furthermore, we note that our operations obey distributive laws, i.e., we have "$a \cdot (b + c) = a \cdot b + a \cdot c = ab + ac$" and "$(a + b) \cdot c = a \cdot c + b \cdot c = ac + bc$".[3]

A careful reader may have noticed that until now we have only considered some parts of an FSM for our matrix representation, namely the states and the edges with their labels. We did not consider how to denote initial and final states. Now, that we have neutral elements, we can close this gap.

1. Initial states are modelled by a line vector solely consisting of 0s and exactly one 1. In particular, if i is the initial state, then the ith entry equals 1.

2. Final states are given by a column vector consisting of 0s and 1s. More precisely, if f is a final state, then the fth entry equals 1. Thus, the final state vector has exactly as many entries equal to 1 than there are final states in the FSM.

For our example FSM in Figure 2.1 the initial state vector is $S = (1, 0)$ while the final state vector is $F = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

Now that we have introduced matrix representations of FSMs, what do we gain from this new representation? The graphical representation, like that in Figure 2.1, is very intuitive for modeling state transitions of systems. The matrix representation introduces a lot of mathematical notation which does not look useful at first sight. Our next goal is to show how matrix representations can be employed to get hold of state transitions via matrix operations, and finally how we can even "calculate" the overall system output in an algebraical and "automated" way.

2.2 State Transitions

At first, note that matrix M of an FSM is a compact representation of all possible state transitions. If we are interested into which state the system may evolve if we know that it currently is in state s, we may, of course, have a look at a graphical FSM representation to find all possible successors of state s, i.e., all states that can be directly reached via an oriented edge rooted at s. However, this can also be achieved by looking at line s of matrix M. All non-zero entries of this line constitute successors of state s. More precisely, if the (s, t) (line s, column t) entry of M is non-zero, then state t is a possible successor of s and if the system evolves along that edge, the output it generates is exactly the label that can be found at entry (s, t) of the matrix.

Consider our example (Figure 2.1) and assume that the system it describes currently is in State 1. Then Figure 2.1 tells us that there are two possible transitions: $(1 \to 1)$ and $(1 \to 2)$. The corresponding output obviously will be "a" or "b", respectively. Having a look at line 1 of the corresponding matrix

$$M = \begin{pmatrix} a & b \\ 0 & a \end{pmatrix}$$

we can derive the same information immediately.

Now, assume that we want to know, what the cumulative output is after two state transitions. Or three transitions. Or four. Or after 42 transitions. This is something that can be found only by very tedious work from the graphical representation of the FSM. As we shall see shortly, such questions can be answered easily with help of the matrix representation of FSMs.

[1] Thus the zero entries in our matrices do make more sense.
[2] From a programmer's perspective and her if-statement analogy this perfectly makes sense.
[3] Which again, from a programmer's point of view, coincides with what she assumes "natural" for if-statements.

In fact, performing k state transitions of FSMs is related to computing the kth power of the corresponding matrix. In order to be able to compute M^k for matrix M, however, we have to know, how to multiply such matrices. So, in our example the task is to compute

$$\begin{pmatrix} a & b \\ 0 & a \end{pmatrix} \cdot \begin{pmatrix} a & b \\ 0 & a \end{pmatrix}.$$

If the reader is familiar with matrix multiplication of matrices that are composed of numbers, she will hardly find something new. The idea behind multiplying our matrices is the same. In order to compute entry (i, j) of the result matrix, we take the ith line of the left matrix and the jth column of the right matrix and multiply the entries of these vectors one by one. We gain the final result, i.e., the entry (i, j), by adding up all of these products. The only difference to numerical calculation is that now we are applying our own "$\cdot$"-operation to calculate the product of the involved entries and we use our own "+"-operation to sum up all the products.

For an appetizer consider we want to calculate entry $(1, 1)$ of the above matrix product. So, we have to take line 1 of the left matrix. This gives (a, b). And we have to take column 1 of the right matrix. This is $\begin{pmatrix} a \\ 0 \end{pmatrix}$. According to the "rules of multiplication" we compute

$$a \cdot a + b \cdot 0 = a^2 + 0 = a^2.$$

Thus we have found the first entry of the result. The remaining three entries can also be calculated with little effort and the reader will easily verify

$$M^2 = M \cdot M = \begin{pmatrix} a & b \\ 0 & a \end{pmatrix} \cdot \begin{pmatrix} a & b \\ 0 & a \end{pmatrix} = \begin{pmatrix} a^2 & ab + ba \\ 0 & a^2 \end{pmatrix}.$$

Note however that entry $(1, 2)$ of the result, $ab + ba$, cannot be simplified any further because juxtaposition, our "$\cdot$"-operation, does not commute. Thus $ab \neq ba$ and $ab + ba \neq ab + ab = ab$.[4]

Now, having a look at line 1 of the result matrix and assuming that the system initially has been in State 1, we see that after two transitions the system either still is in State 1 (by performing the two transitions $1 \to 1 \to 1$ and producing the output $a \cdot a = aa = a^2$) or it has evolved to State 2. In the latter case there are two possible transitions for the system:

$1 \to 1 \to 2$ by producing the output $a \cdot b = ab$ or

$1 \to 2 \to 2$ by generating the output $b \cdot a = ba$.

However, which transitions exactly are performed cannot be seen from matrix M^2. Anyway, we are given the exact output of all transitions.

We can now easily calculate higher powers of M like

$$M^3 = M^2 \cdot M = \begin{pmatrix} a^2 & ab + ba \\ 0 & a^2 \end{pmatrix} \cdot \begin{pmatrix} a & b \\ 0 & a \end{pmatrix} = \begin{pmatrix} a^3 & a^2 b + aba + ba^2 \\ 0 & a^3 \end{pmatrix}$$

and

$$\begin{pmatrix} a & b \\ 0 & a \end{pmatrix}^4 = \begin{pmatrix} a^4 & a^3 b + a^2 ba + aba^2 + ba^3 \\ 0 & a^4 \end{pmatrix}.$$

The reader will find it straightforward to verify that those matrices really represent the output of the system after three resp. four transitions.

[4] However, the simplification $ab + ab = ab$ is correct because we have assumed that "+" is an idempotent operation.

We will now face the more challenging task to compute M^k, i.e., the output of the system after k transitions. With the above results for M^2, M^3, and M^4 in mind, we may guess that

$$\begin{pmatrix} a & b \\ 0 & a \end{pmatrix}^k = \begin{pmatrix} a^k & \sum_{i=0}^{k-1} a^i b a^{k-i-1} \\ 0 & a^k \end{pmatrix}. \tag{2.1}$$

Such formulas can be proved via *mathematical induction*. The general proof idea is the following:

1. Show that the formula is correct for $k = 1$. This is the so-called *base case*.
2. Assume that the formula is correct for $n = 1, \ldots, k$. Prove that under this assumption, it is correct for $n = k+1$. This is called *induction step*.

If this can be proved, the formula is correct for all $k \in \mathbb{N}$, i.e., for all natural numbers k.

To settle the base case for Formula (2.1) is easy. In fact we have already shown that it holds for $k = 1, 2, 3, 4$.

The induction step is more elaborate. We assume that Formula (2.1) is correct for $n = 1, \ldots, k$. Then we have to prove that it is correct for $n = k + 1$, too. For $n = k + 1$ we have to compute

$$M^{k+1} = M^k \cdot M = \begin{pmatrix} a^k & \sum_{i=0}^{k-1} a^i b a^{k-i-1} \\ 0 & a^k \end{pmatrix} \cdot \begin{pmatrix} a & b \\ 0 & a \end{pmatrix}.$$

By simple calculations it is easy to see that

$$M^{k+1} = \begin{pmatrix} a^{k+1} & S_{k+1} \\ 0 & a^{k+1} \end{pmatrix}$$

where

$$S_{k+1} = a^k \cdot b + S_k \cdot a$$

and

$$S_k = \sum_{i=0}^{k-1} a^i b a^{k-i-1}$$

from the case $n = k$.

Thus it remains to show that S_{k+1} coincides with $\sum_{i=0}^{k} a^i b a^{k-i}$ which we derive from Formula (2.1) by incrementing k by one. By performing the following calculations we again have to obey the rules for our "$\cdot$"- and "$+$"-operations.

We compute

$$\begin{aligned} S_{k+1} =& a^k \cdot b + S_k \cdot a = a^k \cdot b + \left(\sum_{i=0}^{k-1} a^i b a^{k-i-1} \right) \cdot a = \\ & a^k \cdot b + \sum_{i=0}^{k-1} a^i b a^{k-i-1} \cdot a = a^k \cdot b + \sum_{i=0}^{k-1} a^i b a^{k-i} = \\ & \sum_{i=0}^{k-1} a^i b a^{k-i} + a^k b = \sum_{i=0}^{k} a^i b a^{k-i}. \end{aligned}$$

This completes the proof that Formula (2.1) is correct for all $k \geq 1$.

Having achieved this result is impressing. However, what is even more impressing, is that it has been done almost automatically. Proofs like that we did above, the mathematical induction proof, can in fact be automated in a straightforward way. Even more impressing is that we can do more! We can derive a formula for all possible outputs of our state transition system!

How could we try to achieve such a goal? What kind of descriptions can we expect for all possible outputs of state transition systems?

First, how could we define "all possible outputs of a state transition system"?

We have seen above that we are able to calculate the output of a system after k transitions for all $k \geq 1$. "All possible outputs" then could mean the union of the output of the system for $k \geq 0$. Now "union" is nothing else than "summing up" our formulas for $k \geq 0$. Thus "all possible outputs" is the same as

$$\sum_{k\geq 0} M^k.$$

Before we can do such a calculation for matrices, we need to know what $\sum_{k\geq 0} a^k$ means for edge labels a. The formula

$$\sum_{k\geq 0} a^k = a^0 + a^1 + a^2 + a^3 + a^4 + \ldots$$

tells us that we can have zero "a"s or one "a" or two "a"s or .... Said differently, we can have an arbitrary number of "a"s. This is commonly written "a^*", i.e.,

$$a^* = \sum_{k\geq 0} a^k$$

and called the "Kleene star". Programmers my interpret "a^*" as a loop with an unknown number of iterations.

Equipped with the star-notation we are now prepared to calculate

$$M^* = \sum_{k\geq 0} M^k.$$

We find

$$M^* = \sum_{k\geq 0} M^k = \begin{pmatrix} \sum_{k\geq 0} a^k & \sum_{k\geq 1}\sum_{i=0}^{k-1} a^i b a^{k-i-1} \\ 0 & \sum_{k\geq 0} a^k \end{pmatrix}.$$

The two $\sum_{k\geq 0} a^k$ can safely be replaced with a^*. The remaining entry at $(1,2)$ is a little bit more complex but can also be simplified by changing the domain of the summation. We obtain

$$\sum_{k\geq 1}\sum_{i=0}^{k-1} a^i b a^{k-i-1} = \sum_{i\geq 0}\sum_{k\geq i+1} a^i b a^{k-i-1} =$$
$$\sum_{i\geq 0} a^i b \sum_{k\geq i+1} a^{k-(i+1)} = \sum_{i\geq 0} a^i b \sum_{k\geq 0} a^k =$$
$$\left(\sum_{i\geq 0} a^i b\right) a^* = \left(\sum_{i\geq 0} a^i\right) b a^* = a^* b a^*$$

Summing up, we found

$$M^* = \begin{pmatrix} a^* & a^*ba^* \\ 0 & a^* \end{pmatrix}.$$

We still have not taken into account the initial and the final states. Remember that we are interested in the output of the system that is generated when the system is started in its initial state and terminated in one of its final states.

These, however, can be taken into account very easily. We only have to compute

$$S \cdot M^* \cdot F$$

where S and F are the initial and final vectors defined above.

In our particular example we obtain

$$S \cdot M^* \cdot F = \begin{pmatrix} 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} a^* & a^*ba^* \\ 0 & a^* \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = a^*ba^*.$$

This means that our transition system will output an arbitrary number of "a"s followed by one single "b" which is followed by an arbitrary number of "a"s again. This is exactly what one would have expected by looking at the graphical representation in Figure 2.1. It has, however, been found in a straight-forward and algebraical manner. Even more important: such computations can be automated easily.

This section has shown that matrix representations of FSMs, though at first sight more complicated and not so intuitive as graphical representations, have advantages. Matrix representations allow for calculating the overall output of state transition system (modelled by FSMs) in a purely algebraical way that is well-suited for automation.[5]

The only matrix operation we needed in this section is matrix multiplication. As we will see in the following chapters, other well-known matrix operations exist that can be employed to perform even more interesting analyses of transition systems and computer programs. In the end, our investigations will allow us to verify concurrently executing threads which are synchronizing their behavior via semaphores or monitors. All of these matrix operations are based on Kronecker product to be introduced in the next chapter.

Exercises

1. Let $\mathcal{L}$ be the set of edge labels of an FSM. Show that $\langle \mathcal{L}, \cdot, + \rangle$, i.e., the labels together with our "$\cdot$"- and "$+$"-operations, form an *idempotent semiring* (also known as a *dioid*).

2. Let $\mathcal{L}$ be the set of edge labels of an FSM. $\langle \mathcal{L}, \cdot, +, * \rangle$, i.e., the labels together with our "$\cdot$", "$+$", and "$*$"-operations, form a *Kleene algebra* and a *complete star semiring*. Show that

   $$(a+b)^* = (a^*b)^*a^* \quad \text{and}$$
   $$(ab)^* = 1 + a(ab)^*b$$

   hold. In addition, consider the equation

   $$x = 1 + a \cdot x. \tag{2.2}$$

   Over the field of real numbers its solution would simply be

   $$x = \frac{1}{1-a}.$$

   In our case, however, we neither have subtraction nor division.

   On the other hand, prove that equation (2.2) is solved by a^*. Thus a^* may be termed *quasi inverse element* of $a \in \mathcal{L}$.

[5] Our informal presentation follows the formal treatment in [KS86].

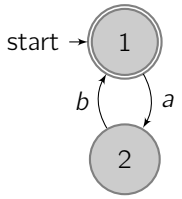


Figure 2.2: FSM F_1

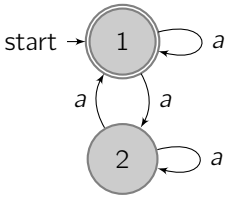


Figure 2.3: FSM F_2

3. Let $\langle \mathcal{L}, \cdot, +, * \rangle$ be the complete star semiring from Exercise 2. In addition, let $\mathcal{M}_n$ be the set of all $n \times n$-matrices with entries from $\mathcal{L}$. Show that $\langle \mathcal{M}_n, \cdot, +, * \rangle$ is a *complete star semiring*. In particular, let I_n be the neutral element of matrix multiplication and Z_n that of matrix addition. How are I_n and Z_n defined?

4. Given the graphical representation of FSM F_1 in Figure 2.2, setup matrix M of its matrix representation and the initial and final state vectors S and F. Compute the first few powers of M in order to conjecture formulas for M^{2k} and M^{2k+1}, $(k \geq 0)$. Prove these formulas via mathematical induction. Finally, find M^* and give the overall output of F!

5. Given the graphical representation of FSM F_2 in Figure 2.3, give the overall output of F by the same lines as in Exercise 4.

Chapter 3

Marching in Lockstep

In this chapter we will give a formal Definition of Kronecker product. Kronecker product, being a matrix operation, also has an interpretation for FSMs. We will also concentrate on this interpretation and provide several examples. In addition, we will show how Kronecker product can be used to verify properties of modular software.

3.1 The Kronecker Product

Kronecker product is also being referred to as *direct product* or *Zehfuss product*. Knuth notes in [Knu11] that Kronecker never published anything about it. Zehfuss[1] was actually the first publishing about this product in the 19th century [Zeh58]. He proved that

$$\det(A \otimes B) = \det^n(A) \cdot \det^m(B) \tag{3.1}$$

if A and B are matrices of order m and n, respectively, and entries from the domain of real numbers. In fact, Zehfuss only gave an example of equation (3.1). The proof of the statement was left to the reader.[2]

The first paper using the name "Kronecker product" seems to be [Wei62].

Definition 1. *Given an m-by-n matrix A and a p-by-q matrix B, their Kronecker product denoted by $A \otimes B$ is a mp-by-nq block matrix defined by*

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

For example, if

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$$

and

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix},$$

[1] In 1858 Dr. Georg Zehfuss was "provisorischer Lehrer der höheren Mathematik an der höheren Gewerbschule zu Darmstadt", which translates into "provisional teacher of higher mathematics at the vocational school at Darmstadt (Hesse/Germany)".

[2] We will frequently rely on that style of proof in this booklet ☺.

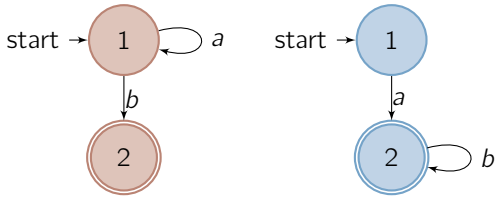


Figure 3.1: FSMs A (left) and B (right)

then

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

The definition of Kronecker product does not tell us what it means to the underlying FSMs when their matrices are Kronecker multiplied. To get some insight we study a simple example. Consider the two FSMs depicted in Figure 3.1. In matrix representation we obtain matrices

$$A = \begin{pmatrix} a & b \\ 0 & 0 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 0 & a \\ 0 & b \end{pmatrix}.$$

The Kronecker product of A and B is computed to be

$$A \otimes B = \begin{pmatrix} \cdot & aa & \cdot & ba \\ \cdot & ab & \cdot & bb \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \tag{3.2}$$

We haven't mentioned yet how initial and final states of the product FSM can be computed. But this is simple: Let S_A and S_B be the initial state vectors of the operands, F_A and F_B their final state vectors. Then the initial vector of the product is given by $S_A \otimes S_B$ and its final vector is $F_A \otimes F_B$.

In our example we have $S_A = (1,0)$, $S_B = (1,0)$, $F_A = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, and $F_B = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Thus $S_{A\otimes B} = (1,0,0,0)$ and

$$F_{A\otimes B} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

which simply state that the initial state of the Kronecker product FSM $A \otimes B$ is state 1 and its final state is state 4.

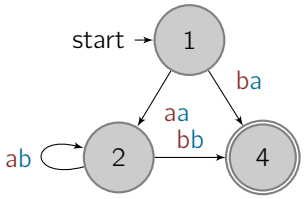


Figure 3.2: Graphical Representation of $A \otimes B$

3.2 Kronecker Product for FSMs

The next question is how Kronecker product can be interpreted on FSM level.

Continuing our example from the previous section, we have now completed all information such that we can give a graphical representation of $A \otimes B$. We start with the initial state 1 and find its successors from the matrix in equation (3.2). These are states 2 and 4. The corresponding edge labels are aa and ba, respectively. Proceeding, we find the successors of state 2 to be state 2 and state 4. State 4, the final state, appears to have no successors. The graphical representation can be seen in Figure 3.2.

There is a surprise in the graphical representation: Although the matrix in equation (3.2) has size four, there are only three states in Figure 3.2. State 3 simply cannot be reached from the initial state 1!

How can we describe what Kronecker product does on FSM level? A short reflection shows that this is very simple: Both FSMs, A and B perform their state transitions in *lockstep*.

1. At the beginning, both FSMs are in their initial state. B has to proceed to state 2, thereby generating output a. A has two possible successor states:
   (a) A stays in state 1, producing output a. This corresponds to state transition $1 \to 2$ in the product FSM.
   (b) A proceeds to its state 2 (output: b). This corresponds to state transition $1 \to 4$ in the product FSM.
2. If the previous step was 1a, both FSMs can now produce together an arbitrary number of ab-pairs. This corresponds to the transition $2 \to 2$ in the product FSM. When A issues a b, B also has to produce a b. This corresponds to transition $2 \to 4$ in the product FSM. Then both FSMs are in their final states. A cannot do a further state transition and both FSMs and the product FSM terminate.
3. If the previous step was 1b, both FSMs are in their final states. Since A cannot proceed, both FSMs and the product FSM terminate.

We have validated our conjecture that the operand FSMs of Kronecker product are executed in lockstep for the above example. Can we be sure, that this conjecture is true for all pairs of FSMs and their Kronecker product? In fact this conjecture can be proved formally by applying methods from formal language theory and algebraical methods similar to those we met in the previous chapter.

Hence we may state

Theorem 1. *Given two FSMs A and B, their Kronecker product FSM generates the same output than A and B do when they execute in lockstep.* □

We carry on with another example to get to know further properties of Kronecker product. Figure 3.3 shows graphical representations of two FSMs, C and D. Taking a closer look at these FSMs, we see that C is able to terminate only when it has performed an even number of state transitions. On the other hand, D can terminate

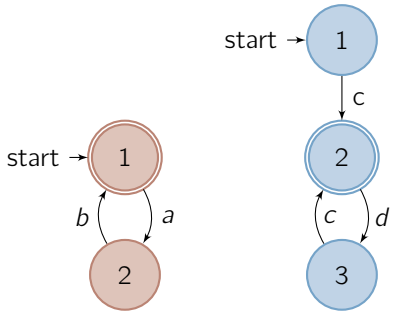


Figure 3.3: FSMs C (left) and D (right)

only after it has done an odd number of transitions. This, of course, is no problem for them as long as we consider the FSMs separately. So, the overall output of C can be found to be $(ab)^*$ and that of D is given by $c(dc)^*$. If, however, we study the Kronecker product of these two FSMs, we encounter a problem. Since FSMs C and D proceed in lockstep, one of them is in a final state if and only if the other is not. For the product FSM this means that it cannot terminate, regardless of how long the FSMs execute. Thus, the product FSM's output equals 0.

We will now see whether we can verify our conjecture via matrix representations. We obtain the matrices

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

and

$$D = \begin{pmatrix} \cdot & c & \cdot \\ \cdot & \cdot & d \\ \cdot & c & \cdot \end{pmatrix}.$$

The Kronecker product of C and D is computed to be

$$C \otimes D = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & ac & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & ad \\ \cdot & \cdot & \cdot & \cdot & ac & \cdot \\ \cdot & bc & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & bd & \cdot & \cdot & \cdot \\ \cdot & bc & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

We continue by constructing a graphical representation of $C \otimes D$ (cf. Figure 3.4). We note that the initial state of $C \otimes D$ is state 1 and its final state is state 2. Although both FSMs are seen to proceed in lockstep producing $ac(bdac)^*$, the final state 2 is not present. State 2 simply cannot be reached from the initial state 1. Thus, we conclude that there is no output of $C \otimes D$ or – said differently – that the overall output of $C \otimes D$ is 0. Hence we have come to the same conclusion as above, but without having to mess around with odd and even numbers of state transitions. The result popped out from pure algebra.

In addition, it is worth noting that – although the matrix $C \otimes D$ has size 6 – only three states appeared in Figure 3.4.[3]

In the above two sections we came to know Kronecker product, which is defined for matrices, and we have seen that the output of the Kronecker product of two FSMs equals the output of the operand FSMs executing in lockstep.

[3]Results on the connectivity of product graphs can e.g. be found in [HIK11, Wei62, McA63, HT66].

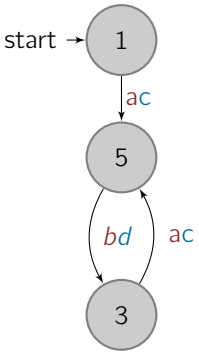


Figure 3.4: Graphical Representation of $C \otimes D$

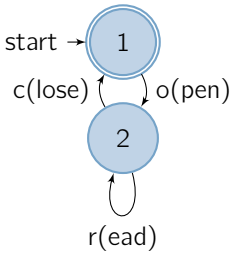


Figure 3.5: Graphical Representation of File Usage Scenario F

The next section will show how Kronecker product can be used to verify some properties of computer programs. In particular, modular programs are composed of building blocks (modules, classes, objects, packages, ...). We will see how such programs can be checked for conforming to the usage patterns defined for the building blocks. Needless to say that Kronecker product plays a major rôle in this process.

3.3 Verifying Programs

Modular software is composed of building blocks, sometimes called modules, classes, objects, or packages. UML, the Unified Modelling Language, provides so-called *state machines* to express usage protocols and to specify the legal usage scenarios of classifiers, interfaces, and ports. These state machines are similar to our FSMs. So, in the following, we do not employ the UML specific diagrams herein, but instead use graphical representations of FSMs to depict UML's state machines.

In addition, we assume that a usage scenario FSM is *deterministic*, which means that for all nodes the outgoing edges have different edge labels[4].

An example specifies simple file handling. Files can be opened and closed. Open files can be used to read data. A graphical representation of that scenario is given in Figure 3.5. From now on, we will use the shorthands "*o*" for *open*, "*c*" for *close*, and "*r*" for *read*.

[4]For each FSM there exists a deterministic version, although this may have more nodes and edges than the indeterministic one.

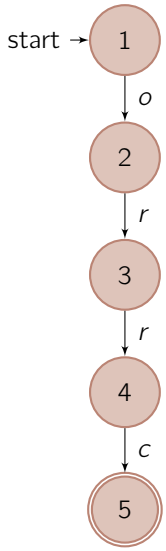


Figure 3.6: Graphical Representation of File Usage System A

A piece of software that uses such a file for implementing a higher-level task, a database management system for example, has to comply to the usage scenario given in Figure 3.5.

In compiler construction it is usual to look at programs not in their source or machine code form, but in a graphical form called *control flow graph* (CFG). Such a CFG consists of basic blocks and of a relation between them which models the flow of control. Graphical representations of CFGs are again very similar to FSMs, but nomenclature changes: instead of states we speak of *nodes*. CFGs have exactly one *root node*, where the execution of the underlying program starts, and it must be guaranteed that there is a *path* between the root node and each of the other nodes. A path is a sequence of nodes such that there is a CFG edge between consecutive nodes of the path. In addition, a CFG has one or more *final nodes* where the execution terminates.

Thus we have a program and a usage scenario both of which can be represented by FSMs. If we want to check whether the program complies to the usage scenario, we could simply execute them in lockstep. If this process terminates, we can be sure that the implementation is correct (at least in this concern).

Returning to our example, a simple program using our file scenario is shown in Figure 3.6. System A opens the file, performs two read operations, and then closes the file. Let's see what happens when we apply Kronecker product to the corresponding matrix representations.

For system A we obtain

$$A = \begin{pmatrix} \cdot & o & \cdot & \cdot & \cdot \\ \cdot & \cdot & r & \cdot & \cdot \\ \cdot & \cdot & \cdot & r & \cdot \\ \cdot & \cdot & \cdot & \cdot & c \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

and for F we get

$$F = \begin{pmatrix} \cdot & o \\ c & r \end{pmatrix}.$$

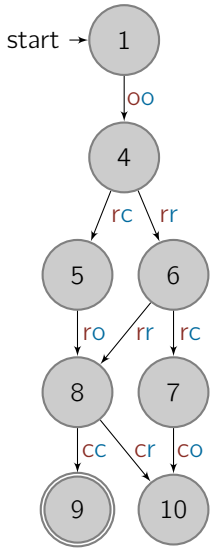


Figure 3.7: Graphical Representation of $A \otimes F$

The Kronecker product $A \otimes F$ is given by

$$A \otimes F = \begin{pmatrix} \cdot & \cdot & \cdot & oo & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & oc & or & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & ro & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & rc & rr & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & ro & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & rc & rr & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & co \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & cc & cr \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

A short calculation shows that node 1 is the root node and node 9 is the final node. So we have gathered enough information to retrieve a graphical representation of $A \otimes F$ which is given in Figure 3.7.

Since we suspected system A to be correct, we expected the overall output to be something like $oorrrrcc$. Although this is contained in Figure 3.7, the graph contains additional nodes and edges. The reason is that not only oo, rr, and cc pairs appear in Figure 3.7. In addition we find rc, ro, ... These pairings, however, do not make sense if we want the program and the usage scenario to be executed in lockstep. Hence, in this context we change the definition of the Kronecker product a little bit: We assume that $rc = ro = \cdots = 0$ and for simplicity set $oo = o$, $rr = r$, and $cc = c$.

This changes result in a new graphical representation of $A \otimes F$ shown in Figure 3.8. We see that Figure 3.6 and Figure 3.8 are very similar. In fact only the node IDs differ.[5]

In graph theory, an *isomorphism* of graphs G and H is a bijection f between the node sets of G and H such that any two nodes u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H. If an isomorphism exists

[5]We tacitly ignore that the two figures have different node colors.

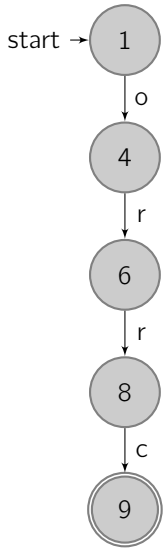


Figure 3.8: Graphical Representation of "new" $A \otimes F$

between two graphs, then the graphs are called *isomorphic*. We write $G \simeq H$ in such a case.

So, clearly the graphs in Figure 3.6 and Figure 3.8 are isomorphic.

Next we consider a system B whose implementation does not comply to the file usage scenario F. Let's assume that the programmer forgets to close the file. So the corresponding FSM looks like that in Figure 3.9.

For system B we obtain

$$B = \begin{pmatrix} \cdot & o & \cdot & \cdot \\ \cdot & \cdot & r & \cdot \\ \cdot & \cdot & \cdot & r \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Matrix F is the same as above and the Kronecker product $B \otimes F$ is calculated to

$$B \otimes F = \begin{pmatrix} \cdot & \cdot & \cdot & o & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & r & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & r \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

A short calculation shows that node 1 is the root node and node 7 is the final node. Thus we obtain the graphical representation of $B \otimes F$ given in Figure 3.10. Curious: Although we suspected system B to be incorrect, we obtain that graph $B \otimes F$ in Figure 3.10 is isomorphic to graph B in Figure 3.9.

Hold on! We were talking of graph isomorphism, but here we are concerned with *control flow graphs*. These are different! So, in addition, we have to take into account root and final nodes. A correct definition now reads as follows:

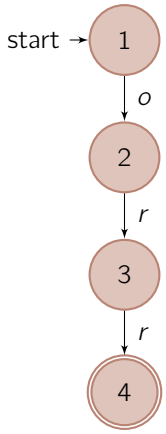


Figure 3.9: Graphical Representation of File Usage System B

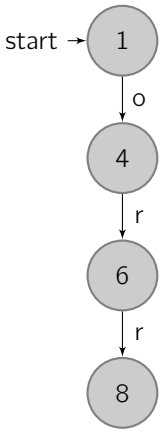


Figure 3.10: Graphical Representation of $B \otimes F$

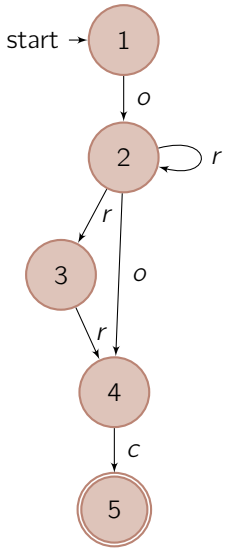


Figure 3.11: Graphical Representation of File Usage System C

Definition 2. *An isomorphism of two control flow graphs G and H is a bijection f between the node sets of G and H such that any two nodes u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H. In addition, let r be the root node of G. Then $f(r)$ has to be the root node of H. For all final nodes s of G, $f(s)$ have to be final nodes of H, and for all final nodes t of H, $f^{-1}(t)$ have to be final nodes in G. If an isomorphism exists between two CFGs, then the CFGs are called* isomorphic *which we denote by $G \simeq H$.*

With this definition we still have $A \otimes F \simeq A$ but $B \otimes F \not\simeq B$ because there is no final node in Figure 3.10.

At last, we try a more complex example. The graphical representation of the CFG of program C is given in Figure 3.11. Here the program opens the file and in a loop reads some data. Then after the loop it does another two read operations before it closes the file. There is, however, an additional edge $(2 \to 4)$ with an open operation, which does not look entirely correct. Anyhow, our calculation gives

$$C \otimes F = \begin{pmatrix} \cdot & o & \cdot & \cdot & \cdot \\ \cdot & r & r & o & \cdot \\ \cdot & \cdot & \cdot & r & \cdot \\ \cdot & \cdot & \cdot & \cdot & c \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} \cdot & o \\ c & r \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & o & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & o & \cdot \\ \cdot & \cdot & \cdot & r & \cdot & r & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & r & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c \end{pmatrix}.$$

The root node of $C \otimes F$ is node 1, the final node is node 9. The corresponding graphical representation can be admired in Figure 3.12. Comparing Figures 3.11 and 3.12 we see that they are not isomorphic, since edge $(2 \to 4)$ of C does not have an adequate counterpart in $C \otimes F$.

In order to setup a general statement, we do the following considerations:

1. Assume we pick a path on program P's side, that complies to the usage scenario U. Then a corresponding path will be present in $P \otimes U$.

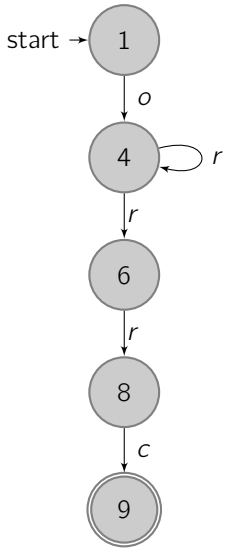


Figure 3.12: Graphical Representation of $C \otimes F$

2. Assume we pick a path on program P's side, that does not comply completely to the usage scenario U. Then a "corresponding" path in $P \otimes U$ will end as soon as the path does not comply to U. This will result in $P \not\simeq P \otimes U$.

Thus we can state:

Theorem 2. *Given P, a CFG of a program, and a usage scenario U, program P complies to U if and only if $P \simeq P \otimes U$.*

This theorem connects "compliance" between a program and a usage scenario to Kronecker product and CFG isomorphism. We already know very well how to deal with Kronecker product. What remains to do, is to have a closer look at the graph isomorphism problem, i.e., we want to know how we can decide whether two CFGs are isomorphic in an algorithmic way. In this regard, it is of critical importance how fast we can decide whether two CFGs are isomorphic. We will deal with the isomorphism problem in the next section.

3.4 Our CFG Isomorphism Problem

Looking up the graph isomorphism problem in text books on algorithmic complexity or in the internet, reveals that this is a very famous problem. At the time being it is one of the rare NP-problems which is neither known to be solvable in polynomial time nor NP-complete. The best current algorithm has time complexity $O(2^{\sqrt{n}\log^2 n})$.

So this is a pity! We have come so far and have very efficient tools at hand for solving software verification. And now we are defeated by the isomorphism.

But, wait! Maybe there is some hope. We do not have to decide on two completely independent graphs. Our graphs are in most cases (very) similar to each other.

Taking a closer look to the "proof" of Theorem 2, which we did immediately before we stated the theorem, we see that we are able to strengthen (2). In fact, when the "corresponding" path in $P \otimes U$ ends, the graph $P \otimes U$ will contain at least one edge less than P. Some other edges and nodes may be missing too, but at least one edge will be omitted. Now, things are getting a lot easier! We can summarize our results:

Theorem 3. *Checking whether $P \simeq P \otimes U$ can be done as follows:*

1. *Check whether the number of nodes and edges in P and $P \otimes U$ are the same.*
   (a) *If "no", we conclude that $P \not\simeq P \otimes U$.*
   (b) *If "yes", continue at 2.*
2. *The root nodes have to present in both CFGs by construction, so we do not have to check these.*
3. *Check whether the number of final nodes in P and $P \otimes U$ is the same.*
   (a) *If "no", we conclude that $P \not\simeq P \otimes U$.*
   (b) *If "yes", we conclude that $P \simeq P \otimes U$, because no additional final nodes are generated by Kronecker product.*

In this section we have studied how programs can be checked whether they comply to the usage scenarios of the modules that are used for implementing the program. However, modules are themselves implemented by other modules. Verifying modules needs slightly different methods. The next section will be concerned with these.

3.5 Verifying Class-like Modules

Modules (classes, packages, ...) differ from ordinary programs in that they consist of several items which are not supposed to execute like programs do. In contrast these items are employed for implementing other modules or programs. Typical examples for such items are the methods of classes and objects or the subroutines being part of Ada packages.

Since in this case we do not have an executable program, how can we check whether the methods interplay in such a way that they will form a correct executable program sometimes later?

Let C denote a class consisting of several methods. We assume that usage scenarios for C have been set up as have been for the modules that have been employed for implementing C's methods. These usage scenarios for C can be considered prototypical implementations of programs using C for their implementation. For this reason, we can check whether this prototypical implementation complies to the usage scenarios of the classes used for implementing the methods of C.

Hence the methods developed for checking programs also apply here. The difference is that we do not check the CFG of a program against usage scenarios, but we check usage scenarios against usage scenarios.

3.6 Inheritance and Usage Scenarios

Object-oriented programming languages provide *inheritance* for building type hierarchies. Several different kinds of inheritance are employed. Here we focus on inheritance as *type extension*.

In this case an FSM of a usage scenario of an inherited class has to be an extension of parent class's usage scenario FSM. Let B be inherited from A. Then we write FSM(B) $\supseteq$ FSM(A). This can be verified easily by checking whether FSM(A) complies to FSM(B).

For *multiple inheritance* let C inherit from both M and N. Then $FSM(C) \supseteq FSM(M)$ and $FSM(C) \supseteq FSM(N)$ and FSM(C) has to comply to both FSM(M) and FSM(N).

Our results by now are impressing: we have very efficient procedures at hand for verifying modular software. They are based on Kronecker product and a very easy decision process for isomorphism of the involved CFGs. An alert reader, however, may have noted that until now we have considered only programs consisting entirely of operations defined in the usage scenario. In a practical setting, programs may and will contain additional statements. We will concentrate on such systems in the next chapter.

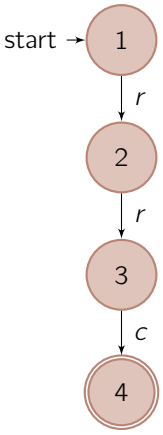


Figure 3.13: Graphical Representation of File Usage System D

Exercises

1. Prove that Kronecker product $\otimes$ is associative, i.e., $(A \otimes B) \otimes C = A \otimes (B \otimes C)$.
2. How is the graph $A \otimes B$ related to the graph $B \otimes A$?
3. Given Program D graphically represented in Figure 3.13, use the method introduced in this chapter to check whether it complies to the usage scenario depicted in Figure 3.5.
4. Given Program E graphically represented in Figure 3.14, use the method introduced in this chapter to check whether it complies to the usage scenario depicted in Figure 3.5.
5. Given Program G graphically represented in Figure 3.15, use the method introduced in this chapter to check whether it complies to the usage scenario depicted in Figure 3.5.
6. Given Program H graphically represented in Figure 3.16, use the method introduced in this chapter to check whether it complies to the usage scenario depicted in Figure 3.5.
7. Given Program J graphically represented in Figure 3.17, use the method introduced in this chapter to check whether it complies to the usage scenario depicted in Figure 3.5.

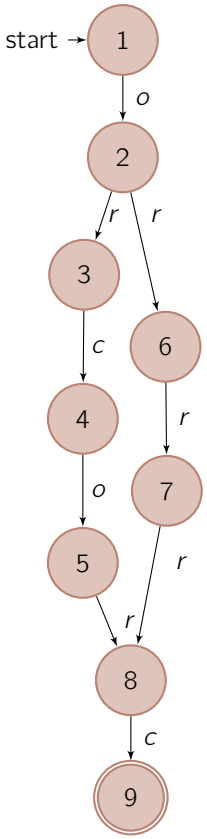


Figure 3.14: Graphical Representation of File Usage System *E*

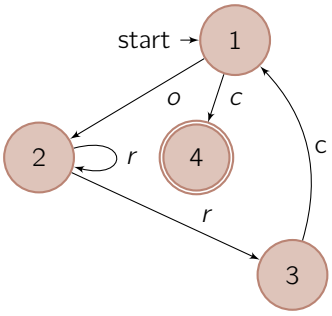


Figure 3.15: Graphical Representation of File Usage System *G*

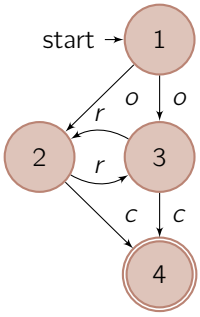


Figure 3.16: Graphical Representation of File Usage System *H*

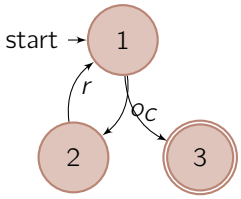


Figure 3.17: Graphical Representation of File Usage System *J*

Chapter 4

Performing Pas de Deux

In the previous chapter we have only considered programs consisting entirely of operations defined in the usage scenario. In this chapter we will study the situation where the program contains additional statements. In contrast to strictly employing Kronecker product for analyzing such programs – as we did in the previous chapter – we now need an operation that allows for "skipping" statements not of interest for the analysis. Anyway, Kronecker product plays a vital rôle in the game.

4.1 The Skip Operation

Consider the following example: Let the file usage scenario be as in the previous chapter (cf. Figure 4.1) and let program K be as shown in Figure 4.2. K consists of the famous file operations plus some additional statements a and b. Strictly applying Kronecker product to the program and the usage scenario does not work in this case because each path containing a or b would make the analysis think that the program is incorrect.

An idea to proceed is to add self loops to all nodes of the usage scenario, where each loop edge label consists of the additional statements $a + b$. Then the analysis should again work properly. But this idea made us alter the file usage scenario which is no good idea. Even worse: We would need a different usage scenario for each program to analyze.

By assuming that the additional statements fulfill $a \cdot a = a$, $b \cdot b = b$, ... and by some mathematical transformations given below, it is, however, possible to remedy this deficiency.

Let $\mathcal{S}$ be the set of operations of usage scenario U, $\mathcal{V}$ a set of additional statements used in program A such that $\mathcal{V} \cap \mathcal{S} = \emptyset$, and let $\mathcal{L} = \mathcal{V} \cup \mathcal{S}$ be set of all operations used in A.

In addition, let $\mathcal{M}_n(\mathcal{L})$ be the set of all matrices of size n with entries $\in \mathcal{L}$. We denote by $I_m \in \mathcal{M}_m$ the identity matrix of size m and by $Z_n \in \mathcal{M}_n$ the zero matrix of size n, i.e., a matrix consisting only of entries of the form 0.

Let $A \in \mathcal{M}_n(\mathcal{L})$ and $U \in \mathcal{M}_m(\mathcal{S})$. In addition we denote the "modified" Kronecker product (skip operation) by $\odot$.

Note that from the matrix' point of view the self loops we have introduced can be modelled by adding a matrix with the only non-zero entries of the form $\sum_{x \in \mathcal{V}} x$ along the main diagonal or – which is the same – by adding an identity matrix multiplied with the scalar $\sum_{x \in \mathcal{V}} x$. Thus

$$A \odot U = A \otimes \left(U + \left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) =$$

$$(A_{\mathcal{V}} + A_{\mathcal{S}}) \otimes \left(U + \left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) =$$

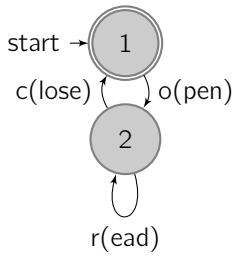


Figure 4.1: Graphical Representation of File Usage Scenario F

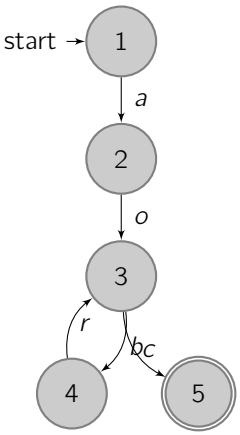


Figure 4.2: Graphical Representation of File Usage System K

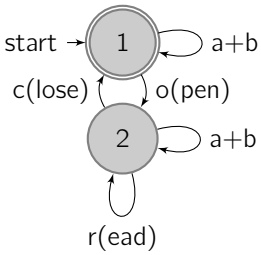


Figure 4.3: Graphical Representation of Modified File Usage Scenario F

(where $A_{\mathcal{V}} + A_{\mathcal{S}} = A$, $A_{\mathcal{V}} \in \mathcal{M}_n(\mathcal{V})$, and $A_{\mathcal{S}} \in \mathcal{M}_n(\mathcal{S})$)

$$A_{\mathcal{V}} \otimes \left(U + \left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) + A_{\mathcal{S}} \otimes \left(U + \left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) =$$
$$A_{\mathcal{V}} \otimes U + A_{\mathcal{V}} \otimes \left(\left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) + A_{\mathcal{S}} \otimes U + A_{\mathcal{S}} \otimes \left(\left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) =$$

(Note that $A_{\mathcal{V}} \otimes U = A_{\mathcal{S}} \otimes \left(\left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) = Z_{n \cdot m}$ since the operands of the Kronecker products contain disjoint sets of entries only.)

$$A_{\mathcal{V}} \otimes I_m + A_{\mathcal{S}} \otimes U.$$

Note that in $A_{\mathcal{V}} \otimes \left(\left(\sum_{x \in \mathcal{V}} x \right) \cdot I_m \right) = A_{\mathcal{V}} \otimes I_m$ because for all $x \in \mathcal{V}$ we have $x \cdot x = x$.

We have thus found a simple formula for our skip operations that is build up of two Kronecker products and one "+" operation.

Theorem 4. *With the definitions above, a program A can be checked whether it complies to usage scenario U by calculating*

$$A \odot U = A_{\mathcal{V}} \otimes I_m + A_{\mathcal{S}} \otimes U.$$

4.2 Verifying Modular Programs

To continue our example we setup matrix A

$$K = \begin{pmatrix} \cdot & a & \cdot & \cdot & \cdot \\ \cdot & \cdot & o & \cdot & \cdot \\ \cdot & \cdot & \cdot & b & c \\ \cdot & \cdot & r & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

So we get

$$K_{\{a,b\}} \otimes I_2 = \begin{pmatrix} \cdot & a & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & b & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} 1 & \cdot \\ \cdot & 1 \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

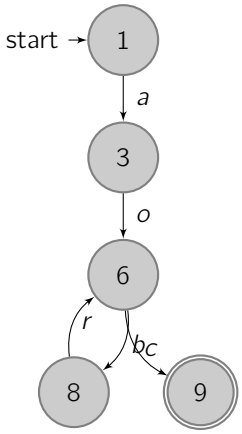


Figure 4.4: Graphical Representation of $K_{\{a,b\}} \otimes I_2 + K_{\{o,r,c\}} \otimes F$

and

$$K_{\{o,r,c\}} \otimes F = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & o & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & c \\ \cdot & \cdot & r & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} \cdot & o \\ c & r \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & o & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & c & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & r & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

By adding we obtain

$$K_{\{a,b\}} \otimes I_2 + K_{\{o,r,c\}} \otimes F = \begin{pmatrix} \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & o & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & c & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & r & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

The initial node is node 1, the final node is node 9. A graphical representation of the resulting graph is shown in Figure 4.4. It is easy to see that the CFG in Figure 4.4 is isomorphic to that in Figure 4.2

4.3 Skeletons

Instead of allowing statements different from those present in the usage scenario, another approach is to eliminate those additional statements. As we will see soon, this works, but it has the disadvantage that debugging incorrect programs may get more difficult because locating the bug in the original program or CFG is not easy.

The first step to eliminate additional statements is to replace the corresponding edge labels in the CFG with "1".[1]

The next step is to eliminate the "1"-edges. The following algorithm does this:

1. If there are no "1"-edges, the algorithm terminates, otherwise go to 2.
2. Remove all "1"-self-loops. If there are no more "1"-edges, the algorithm terminates, otherwise go to 3.
3. Pick a node n with at least one outgoing "1"-edge, $(n \to m)$. Let $\mathrm{Pred}(n)$ denote the set of all predecessors of n. For all edges from a node $p \in \mathrm{Pred}(n)$ to n, add an edge $(p \to m)$. Remove the "1"-edge $(n \to m)$. Do this for all "1"-edges rooted at node n. Finally, go to 1.

Since each time step 3 of the algorithm is executed, the number of nodes with outgoing "1"-edges is decremented by one, the algorithm will terminate.

The final step is that all nodes from which no final node can be reached are deleted from the 1-free graph.

A final note applies to the initial node: The original initial node may be the root node of some "1"-edges. Since these are eliminated, all nodes that can be reached via paths containing only "1"-edges, become initial nodes in the the 1-free graph. Thus, the 1-free CFG does not conform to our definition of a CFG. However, several initial (root) nodes do not pose a real problem. In fact, all our methods and algorithms can be adapted to suit several initial nodes.

We call such a 1-free CFG *usage scenario skeleton*.

As an example consider file usage system R in Figure 4.5. In order to obtain the file usage skeleton, we replace edge labels a, b, d, e, f, g, h, and i with 1 and get the CFG shown in Figure 4.6. According to the algorithm above we choose edge $(1 \to 2)$ to be eliminated and arrive at Figure 4.7. Next we eliminate edge $(3 \to 4)$ (Figure 4.8), $(4 \to 3)$ (Figure 4.9), $(3 \to 5)$ (Figure 4.10), $(5 \to 6)$ (Figure 4.11), $(6 \to 7)$ (Figure 4.12), $(7 \to 5)$ (Figure 4.13), $(5 \to 8)$ (Figure 4.14), and $(9 \to 10)$ (Figure 4.15). After removing nodes 1, 3, 4, 5, 7, and 9, from which the final node 10 cannot be reached, we arrive at the CFG shown in Figure 4.16, the file usage skeleton of system R. The size of the CFG has dropped from ten to four nodes. Thus checking the system against the file usage scenario has become much easier and it clearly turns out that the system complies to the file usage scenario.

We have thus mastered the task of allowing additional statements in the programs to be checked against a given usage scenario. In a practical setting, however, programs may use several objects of the same type (class, ...) at the same time. We will study this problem in the next chapter.

Exercises

1. Given program Q in Figure 4.17, check it against the file usage scenario of Figure 4.1.
2. Change edge label e to c in Figure 4.17 and check the altered program Q' against the file usage scenario of Figure 4.1.
3. Can the skeleton in Figure 4.16 be further simplified?
4. Is it possible that the order in that "1"-edges are eliminated affects the structure of the skeleton?
5. Determine the usage skeleton of program Q and Q'.

[1] The empty string "1" is sometimes also written "λ" or "ε". For this reason automata containing "ε" edge labels are called automata with ε moves and automata without such edge labels are called ε-free or λ-free.

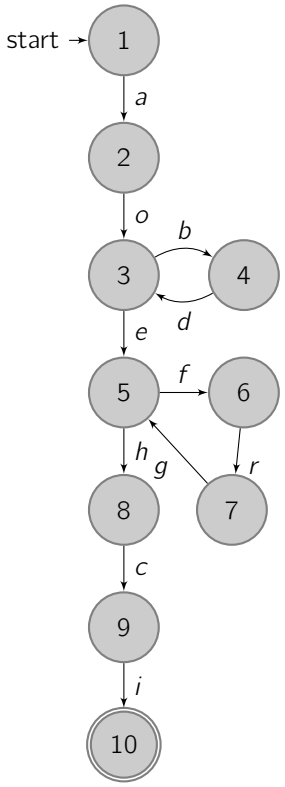


Figure 4.5: Graphical Representation of File Usage System R

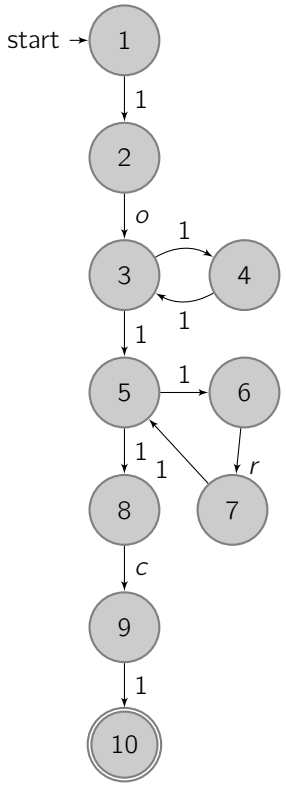


Figure 4.6: Graphical Representation of File Usage System R with "1" Edge Labels

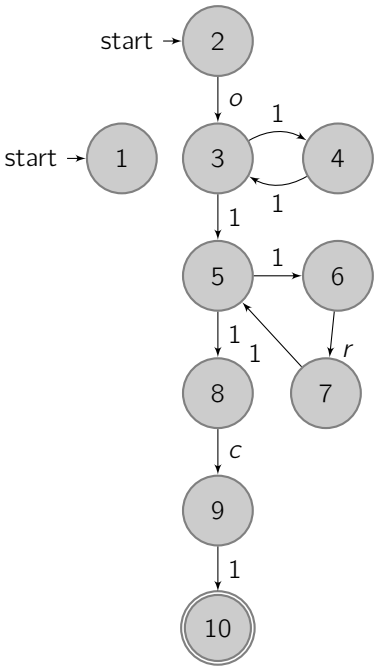


Figure 4.7: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(1 \to 2)$

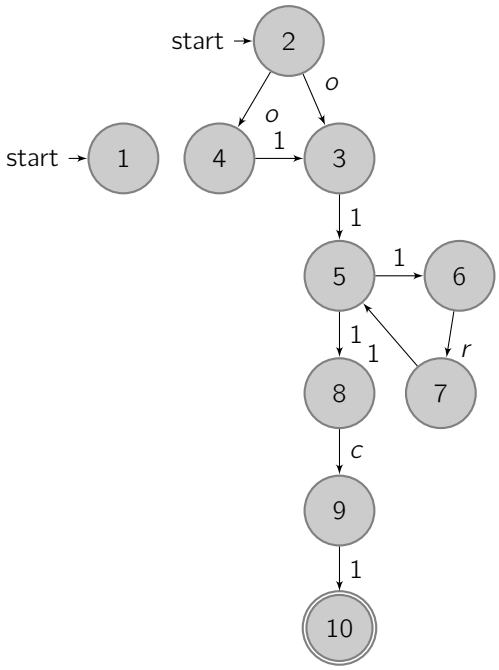


Figure 4.8: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(3 \rightarrow 4)$

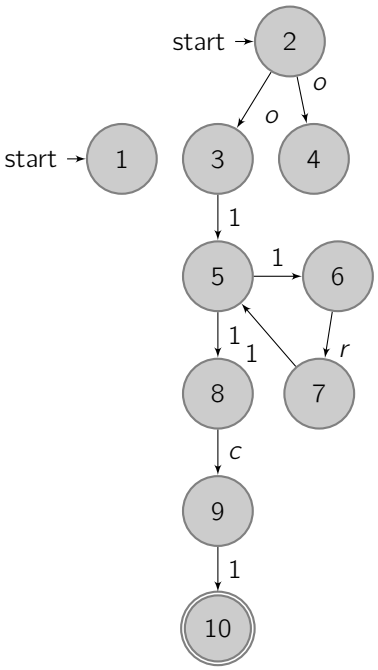


Figure 4.9: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(4 \to 3)$

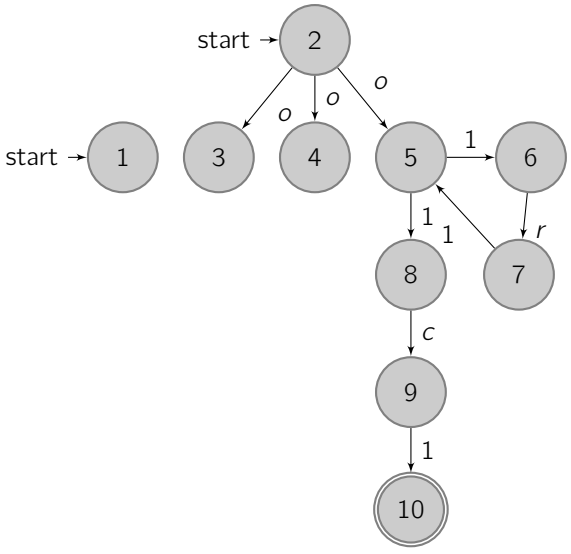


Figure 4.10: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(3 \to 5)$

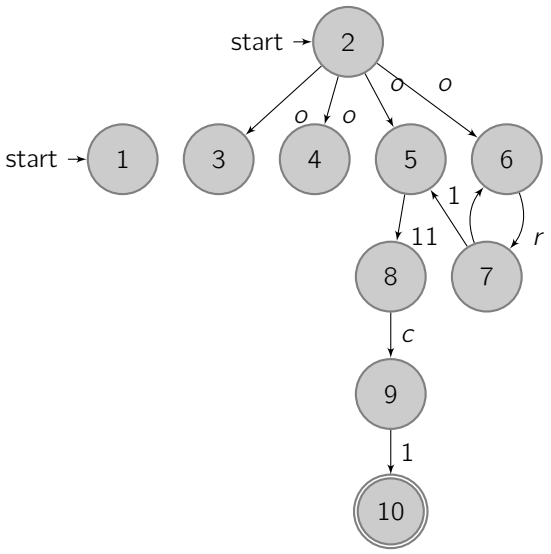


Figure 4.11: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(5 \to 6)$

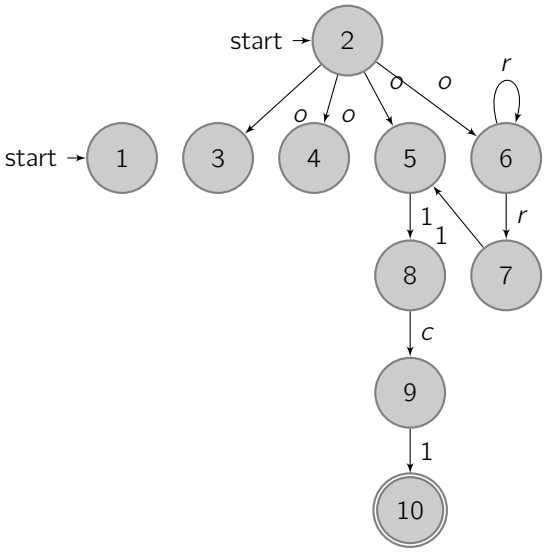


Figure 4.12: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(7 \to 6)$

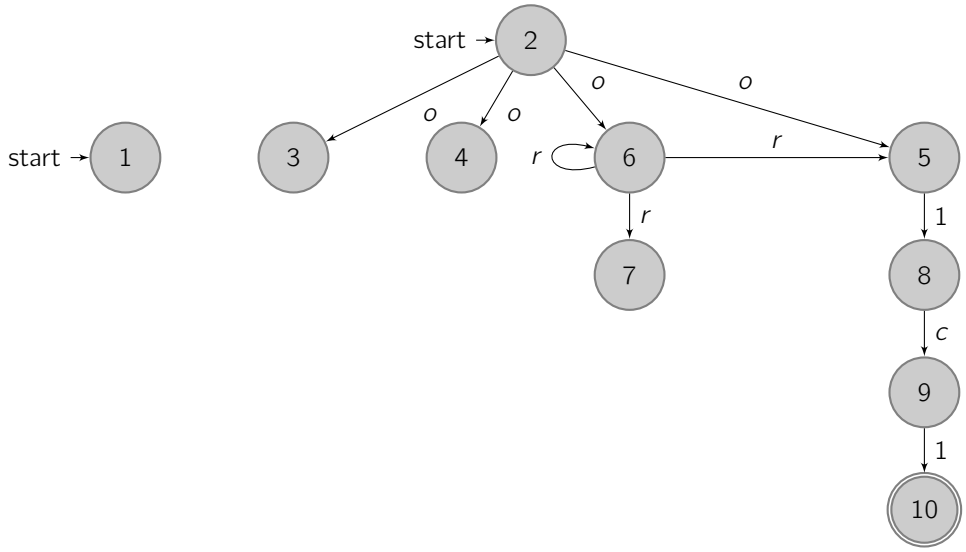


Figure 4.13: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(7 \to 5)$

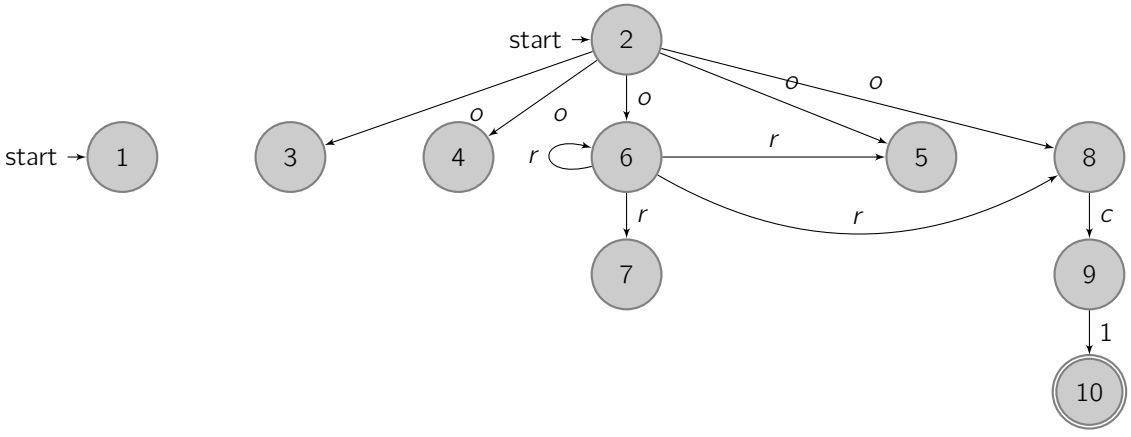


Figure 4.14: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(5 \to 8)$

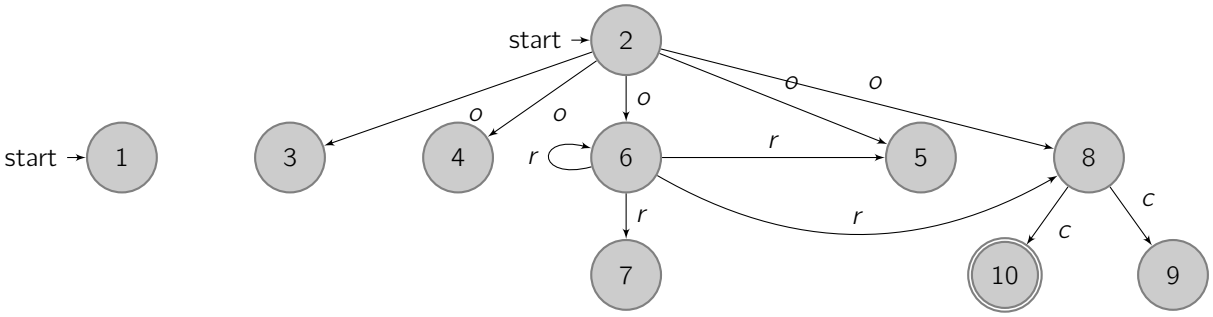


Figure 4.15: Graphical Representation of File Usage System R with "1" Edge Labels After Eliminating Edge $(9 \to 10)$

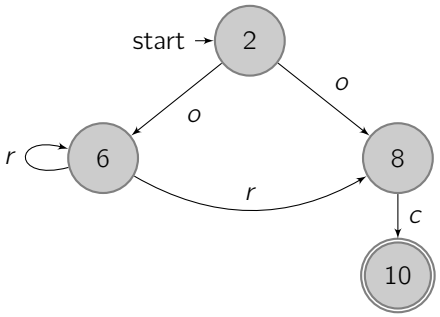


Figure 4.16: Graphical Representation of Usage Skeleton of File System R

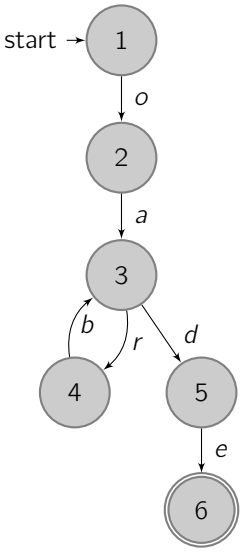


Figure 4.17: Graphical Representation of File Usage System Q

Chapter 5

Ménage à Trois

In practical settings, programs will use several objects (classes, modules, packages, ...) for implementing their own task. Following the lines of the previous two chapters, checking against each of their usage scenarios could be done one after the other. In this section, however, we will show how one can check a program against two and more usage scenarios at the same time. To this end, we have to introduce a new matrix operation, the so-called *Kronecker sum*.

5.1 The Kronecker Sum

Assume that we want to check our program against usage scenarios U and W at the same time. For scenario U this means that in whatever state U may be, it has to accept arbitrary transitions of W. The other way this also holds for all states in W, i.e., arbitrary transitions of U must be accepted in whatever state W is. But this is very similar to what we had in the previous chapter when we wanted to skip operations not being part of the usage scenario.

In the following we assume that $U \in M_n(\mathcal{U})$ and $W \in M_m(\mathcal{W})$ and that $\mathcal{U} \cap \mathcal{W} = \emptyset$. We define Kronecker sum (written $\oplus$) and derive similar to the previous chapter by introducing self loops on both sides:

$$U \oplus W = U \otimes \left(W + \left(\sum_{x \in \mathcal{U}} x \right) I_m \right) + \left(U + \left(\sum_{y \in \mathcal{W}} y \right) I_n \right) \otimes W =$$
$$U \otimes W + U \otimes \left(\left(\sum_{x \in \mathcal{U}} x \right) I_m \right) + U \otimes W + \left(\left(\sum_{y \in \mathcal{W}} y \right) I_n \right) \otimes W.$$

Note that $U \otimes W = Z_{n \cdot m}$ because the edge labels of U and W are disjoint. Hence we get

$$U \oplus W = U \otimes I_m + I_n \otimes W.$$

Thus we have derived the following theorem.

Theorem 5. *With the definitions above, a program A can be checked against two usage scenarios U and W by calculating*

$$A \odot (U \oplus W) = A \odot (U \otimes I_m + I_n \otimes W)\,.$$

We can even check a program against more than two usage scenarios at the same time. In order to be able to write this in a concise way, we introduce

$$\bigoplus_{x \in \mathcal{X}} x$$

similar to the sigma notation for standard sums. Hence we get

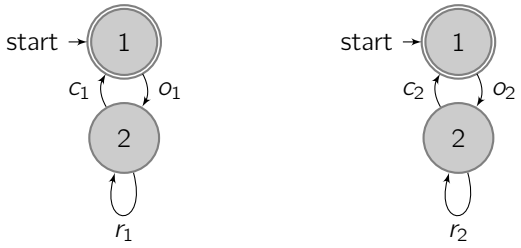


Figure 5.1: Graphical Representation of File Usage Scenarios U and W

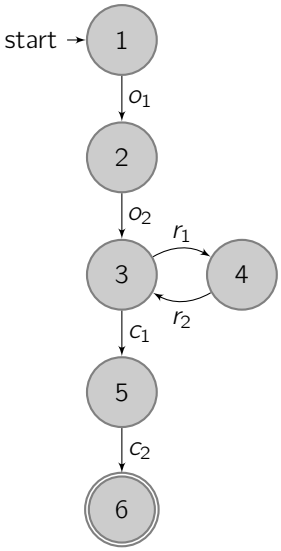


Figure 5.2: CFG of Program A

Theorem 6. *With the definitions above, a program A can be checked against usage scenarios U_i where $i = 1, \ldots, k$ by calculating*

$$A \odot \left(\bigoplus_{i=1}^{k} U_i \right).$$

5.2 One Program – Two Objects

As an example assume we want to check program A against two file objects. The file usage scenarios are shown in Figure 5.1. The CFG of program A is given in Figure 5.2. We obtain the following matrices

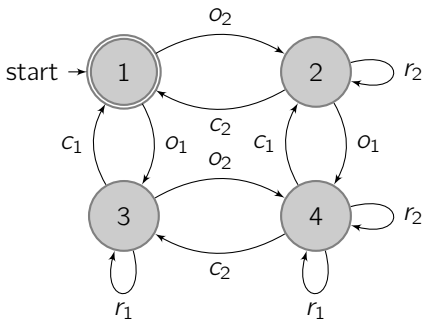


Figure 5.3: Graphical Representation of $U \oplus W$

$$U = \begin{pmatrix} \cdot & o_1 \\ c_1 & r_1 \end{pmatrix},$$
$$W = \begin{pmatrix} \cdot & o_2 \\ c_2 & r_2 \end{pmatrix},$$

and

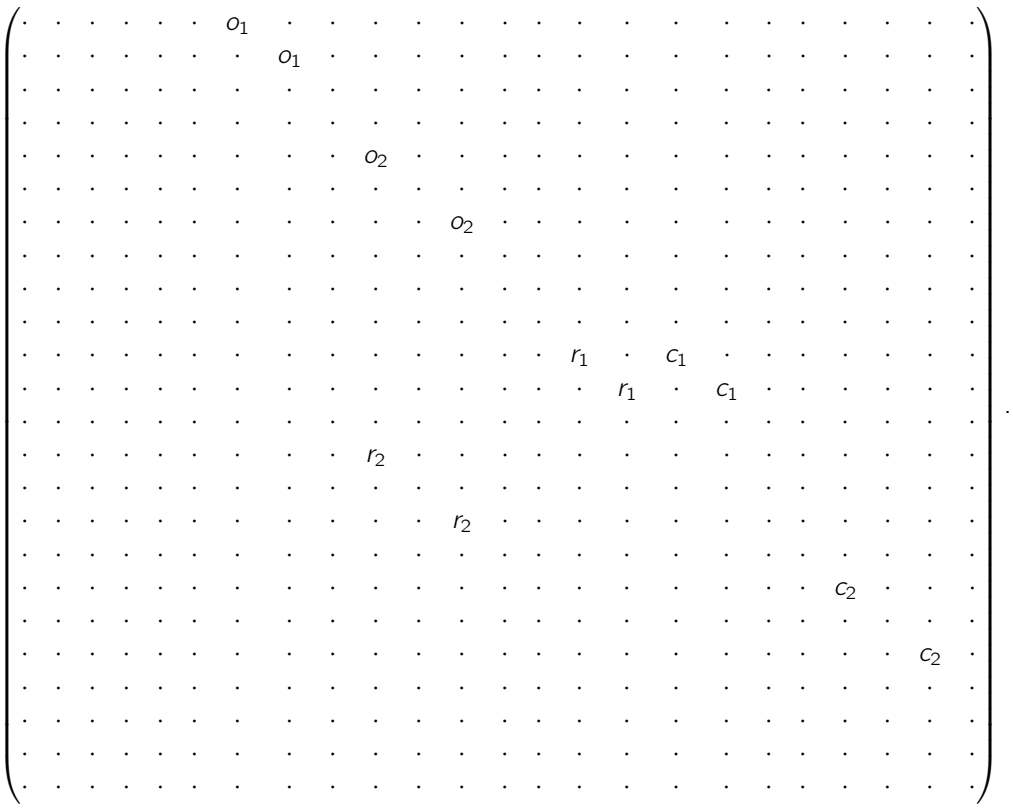
$$A = \begin{pmatrix} \cdot & o_1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & o_2 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & r_1 & c_1 & \cdot \\ \cdot & \cdot & r_2 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & c_2 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Next, we have to compute

$$\begin{aligned} U \oplus W =& U \otimes I_2 + I_2 \otimes W = \\ & \begin{pmatrix} \cdot & o_1 \\ c_1 & r_1 \end{pmatrix} \odot \begin{pmatrix} 1 & \cdot \\ \cdot & 1 \end{pmatrix} + \begin{pmatrix} 1 & \cdot \\ \cdot & 1 \end{pmatrix} \odot \begin{pmatrix} \cdot & o_2 \\ c_2 & r_2 \end{pmatrix} = \\ & \begin{pmatrix} \cdot & \cdot & o_1 & \cdot \\ \cdot & \cdot & \cdot & o_1 \\ c_1 & \cdot & r_1 & \cdot \\ \cdot & c_1 & \cdot & r_1 \end{pmatrix} + \begin{pmatrix} \cdot & o_2 & \cdot & \cdot \\ c_2 & r_2 & \cdot & \cdot \\ \cdot & \cdot & \cdot & o_2 \\ \cdot & \cdot & c_2 & r_2 \end{pmatrix} = \\ & \begin{pmatrix} \cdot & o_2 & o_1 & \cdot \\ c_2 & r_2 & \cdot & o_1 \\ c_1 & \cdot & r_1 & o_2 \\ \cdot & c_1 & c_2 & r_1 + r_2 \end{pmatrix}. \end{aligned}$$

Although, we do not need it in the end, it is illustrative to have a look at the graphical representation of $U \oplus W$. It is depicted in Figure 5.3.

Finally, we compute $A \odot (U \oplus W)$ and get



Since the initial state is node 1 and the final state is node 21, we get the graphical representation of the result in Figure 5.4 which clearly is isomorphic to the CFG in Figure 5.2. Thus we are sure, that the implementation conforms to both usage scenarios.

As a sideline we see that the whole situation gets a little bit confusing if more than two partners are involved since the size of the matrices has increased.

Exercises

1. Prove that Kronecker sum $\oplus$ is an associative operation, i.e., $(A \oplus B) \oplus C = A \oplus (B \oplus C)$.
2. Is there a simple relation between the graphs $(A \odot U) \odot W$ and $A \odot (U \oplus W)$ if A complies to both usage scenarios U and W?
3. Building on exercise 2 what can be said on the graphs $(A \odot W) \odot U$ and $A \odot (W \oplus U)$ when compared to $(A \odot U) \odot W$ and $A \odot (U \oplus W)$?
4. Set up a program that does not comply to the two usage scenarios in Figure 5.1 such that the size of the result matrix is still manageable. Prove your hypothesis!

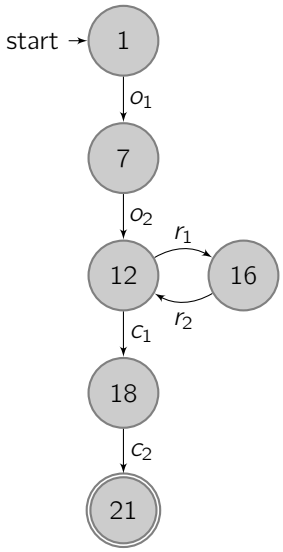


Figure 5.4: Graphical Representation of $A \odot (U \oplus W)$

Chapter 6

Threads without Threats

In this chapter we apply the idea, we used for introducing the skip operation $\odot$ in Chapter 4, to concurrently executing programs, i.e., to several threads of the program executing concurrently. We will show that this can be done even when the threads synchronize their execution via semaphores.

6.1 Concurrent Programs and the Kronecker Sum

In the previous chapter we have defined Kronecker sum to analyze several usage scenarios at the same time. The same idea can be employed to model concurrently executing threads of a program. Assume that the edge labels of two threads T_1 and T_2 are disjoint, and let their CFGs be represented by matrices T_1 and T_2. Then, by the same reasoning as in Chapter 5, the behavior of the concurrent program consisting of threads T_1 and T_2 can be modeled by

$$T_1 \oplus T_2.$$

For example consider the CFGs of thread A and B shown in Figures 6.1 and 6.2. We easily get matrices

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

and

$$B = \begin{pmatrix} \cdot & e & g \\ \cdot & \cdot & f \\ \cdot & \cdot & \cdot \end{pmatrix}.$$

We calculate

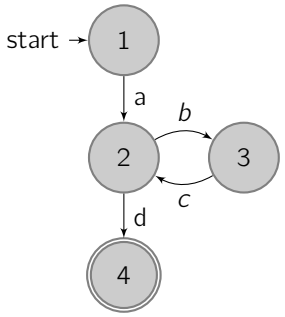


Figure 6.1: CFG of Thread A

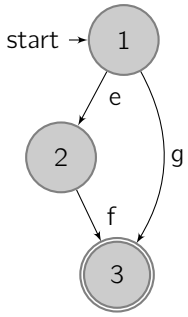


Figure 6.2: CFG of Thread B

$$A \oplus B = A \odot I_3 + I_4 \odot B =$$

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

$$\begin{pmatrix} \cdot & \cdot & \cdot & a & \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 & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & d & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & d & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & d \\ \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 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \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 & g & \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 & g & \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 & \cdot & e & g & \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 & e & g \\ \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} =$$

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

From this matrix we find its graphical representation in Figure 6.3. The reader is invited to check that the graph in Figure 6.3 really models all interleavings that are possible if threads A and B execute concurrently.

However, in practical settings, a concurrent program that consists of completely independent threads, hardly exists. In all relevant cases, the threads of a program somehow communicate or at least synchronize in order to complete their tasks. We will concentrate on synchronization issues in the next section.

6.2 Semaphores

In most cases, threads of a concurrent program communicate with each other. Synchronization may be done by accessing shared resources in a controlled way such that there are no data races, i.e., data is read and/or written at the same time. One means for synchronization of threads are *semaphores*.

In its simplest form, a semaphore can be viewed as a variable that has an integer value upon which three operations are defined [Sta01]:

1. A semaphore may be initialized to a non-negative value.

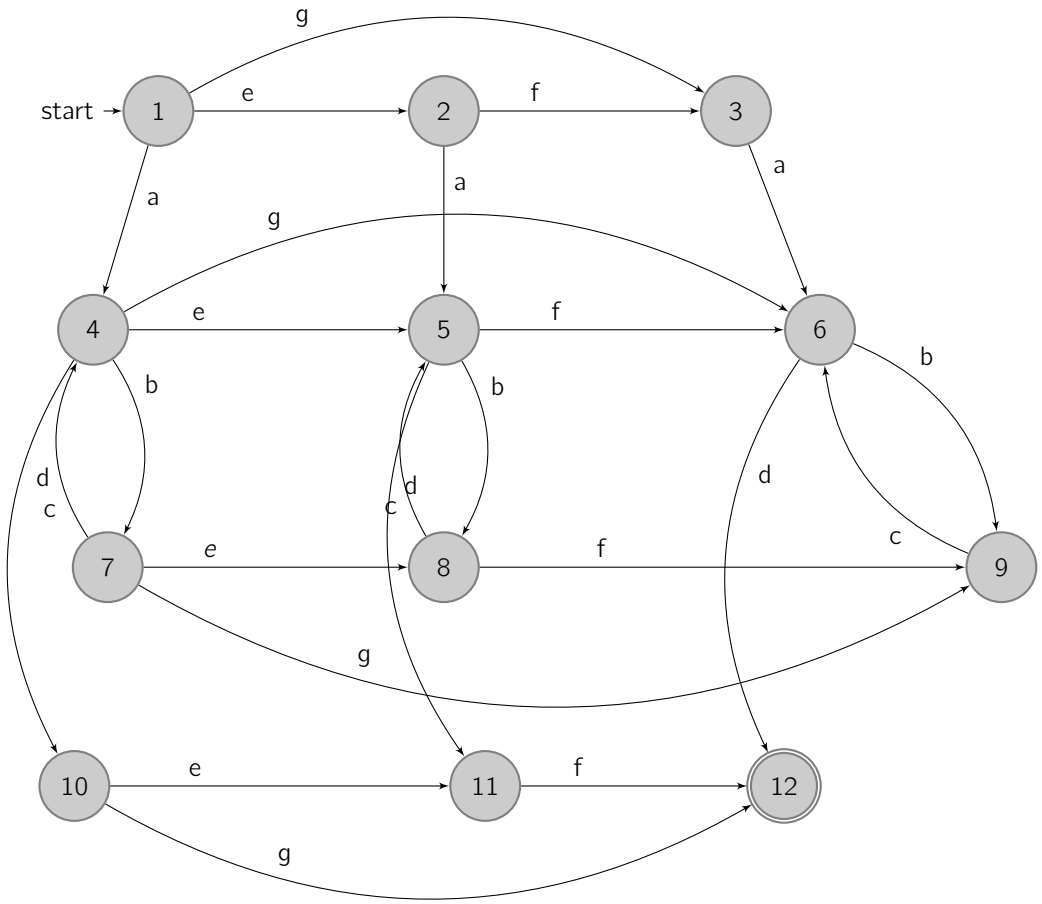


Figure 6.3: Graphical Representation of $A \oplus B$

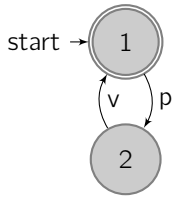


Figure 6.4: CFG of a Binary Semaphore

2. The operation p decrements the semaphore value. If the value becomes negative, then the process (thread) executing operation p is blocked.

3. The operation v increments the semaphore value. If the value is not positive, then a process (thread) blocked by a p operation is unblocked.

Other than these three operations, there is no way to inspect or manipulate semaphores. The operations p and v are assumed to be atomic.

A more restricted version of a semaphore, the *binary semaphore*, may only take on the values 0 and 1. Non-binary semaphores are called *counting semaphores*.

Within the Kronecker model we have a simple representation of a binary semaphore given in Figure 6.4. Since the Kronecker model of concurrent systems is one without time we do not need to worry about waiting queues for threads and other implementation details like *test-and-set-lock* commands.

6.3 Synchronized Multi-threaded Programs and the Kronecker Sum

In the following we assume that $A \in M_n(\mathcal{A})$ and $B \in M_m(\mathcal{B})$ and that $\mathcal{A} \cap \mathcal{B} \subseteq \mathcal{S}$, where $\mathcal{S}$ denotes the set of operations used for synchronization. We check what happens if we apply our trick from the skip operation $\odot$ to threads, i.e., we add self loops on both sides.

$$A \otimes \left(B + \left(\sum_{x \in \mathcal{A}} x \right) I_m \right) + \left(A + \left(\sum_{y \in \mathcal{B}} y \right) I_n \right) \otimes B =$$
$$A \otimes B + A \otimes \left(\left(\sum_{x \in \mathcal{A}} x \right) I_m \right) + A \otimes B + \left(\left(\sum_{y \in \mathcal{B}} y \right) I_n \right) \otimes B$$

In contrast to Section 5, here $A \otimes B \neq Z_{n \cdot m}$ because $\mathcal{A} \cap \mathcal{B}$ may not be empty. But $A \otimes B \neq Z_{n \cdot m}$ can happen only if synchronize operations are executed at the same time by both threads A and B. Since this is not allowed, anyway, we simply remove the $A \otimes B$-terms above to get the following theorem.

Theorem 7. *With the definitions above, a program consisting of two concurrently executing threads A and B and one single semaphore S can be modeled by*

$$(A \oplus B) \odot S = (A \otimes I_m + I_n \otimes B) \odot S.$$

It is easy to generalize the theorem above to a finite number of threads and a finite number of semaphores.

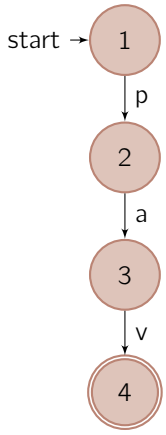


Figure 6.5: CFG of Thread A

Theorem 8. *Let T_i for $i = 1, \ldots, n$ be concurrently executing threads and S_j for $j = 1, \ldots, k$ be semaphores. Then the program consisting of the threads T_i $(i = 1, \ldots, n)$ and the semaphores S_j $(j = 1, \ldots, k)$ can be modeled by*

$$\bigoplus_{i=1}^{n} T_i \odot \bigoplus_{j=1}^{k} S_j.$$

As an example consider two threads A and B with the CFGs presented in Figures 6.5 and 6.6. Both threads are using a semaphore for synchronization issues. The CFG of the semaphore is still that of Figure 6.4. For the matrices we obtain easily

$$A = \begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

and

$$B = \begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & b & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

We compute $A \oplus B$

$$A \oplus B = A \otimes I_4 + I_4 \otimes B =$$
$$\begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{pmatrix} + \begin{pmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{pmatrix} \otimes \begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & b & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} =$$

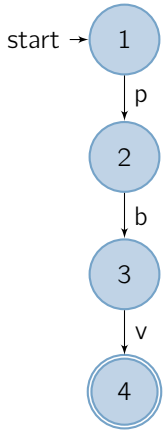


Figure 6.6: CFG of Thread B

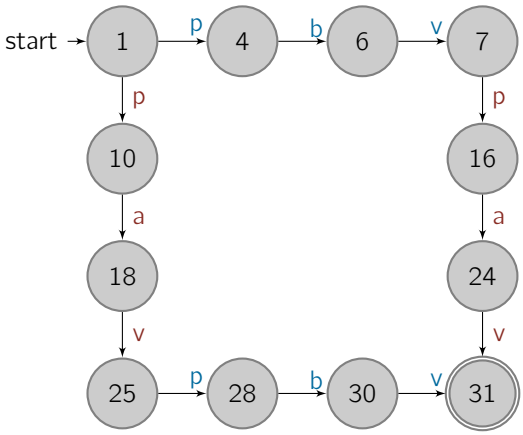
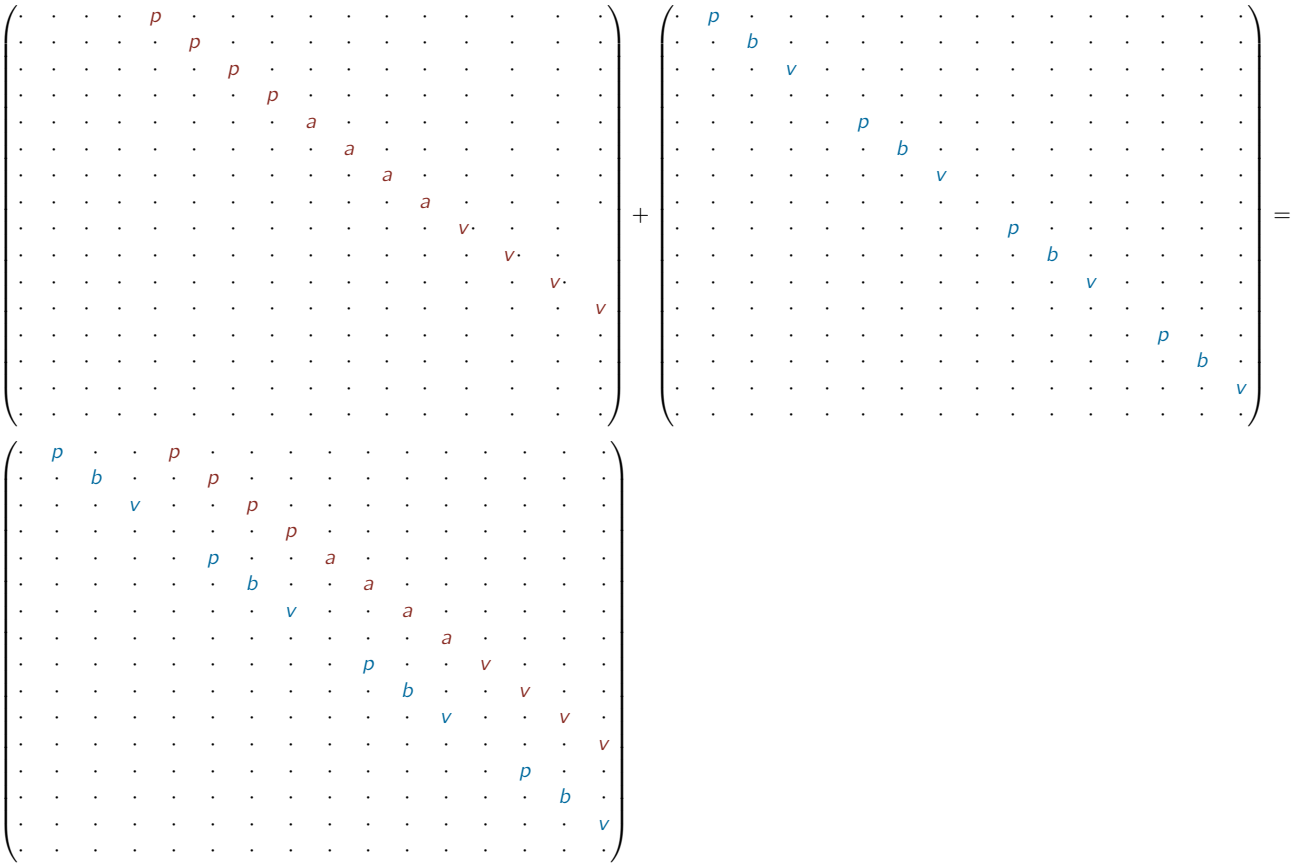
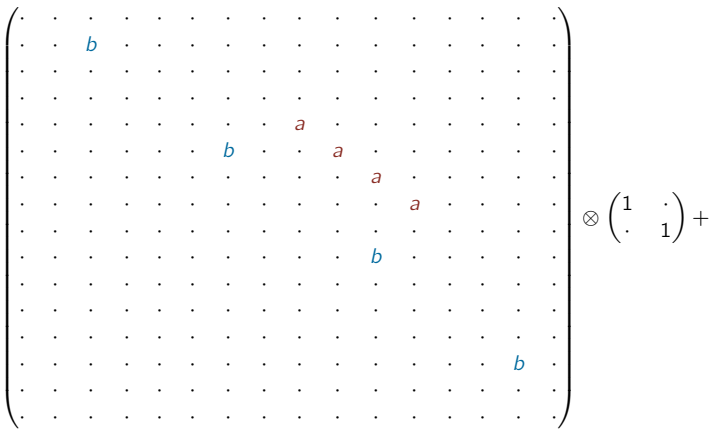
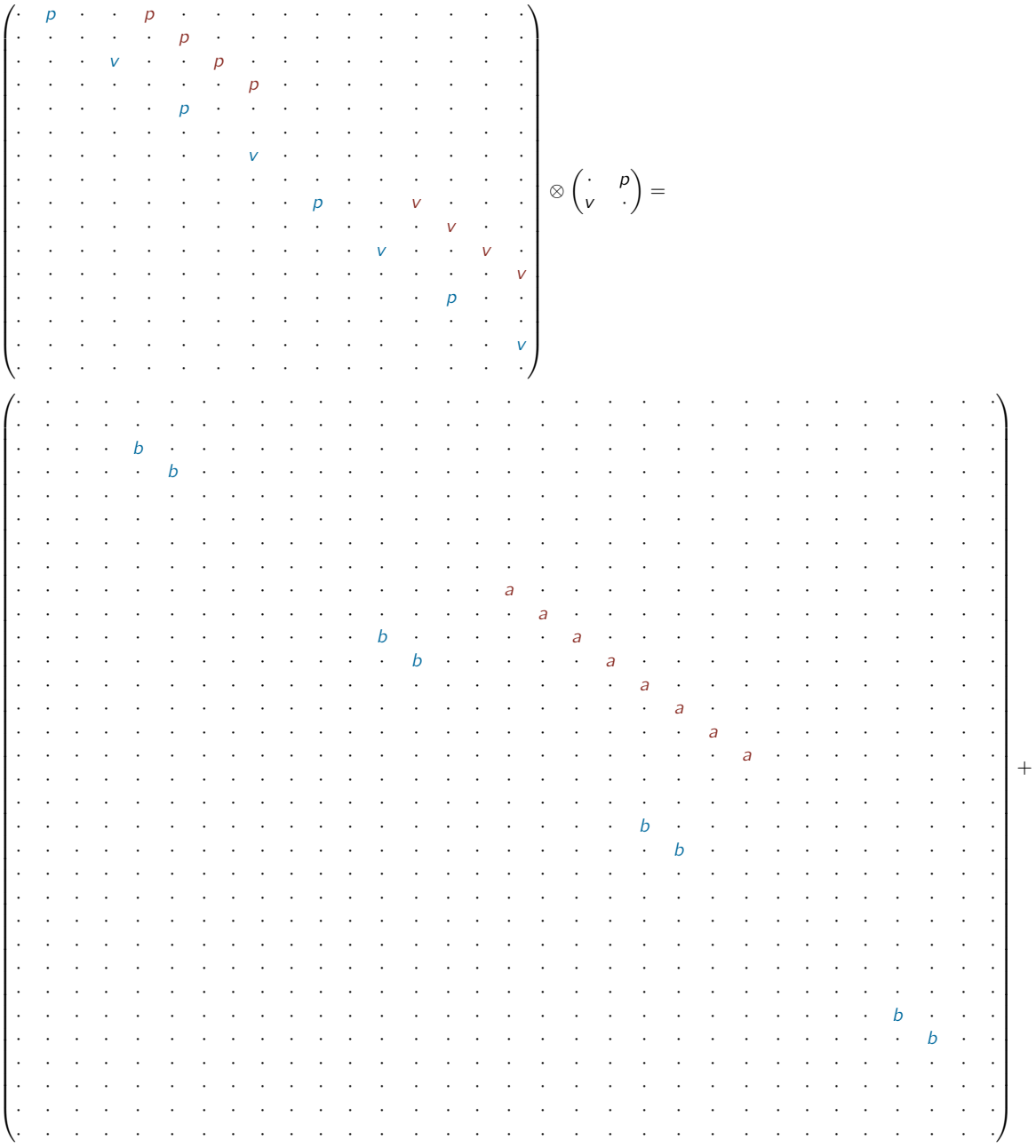


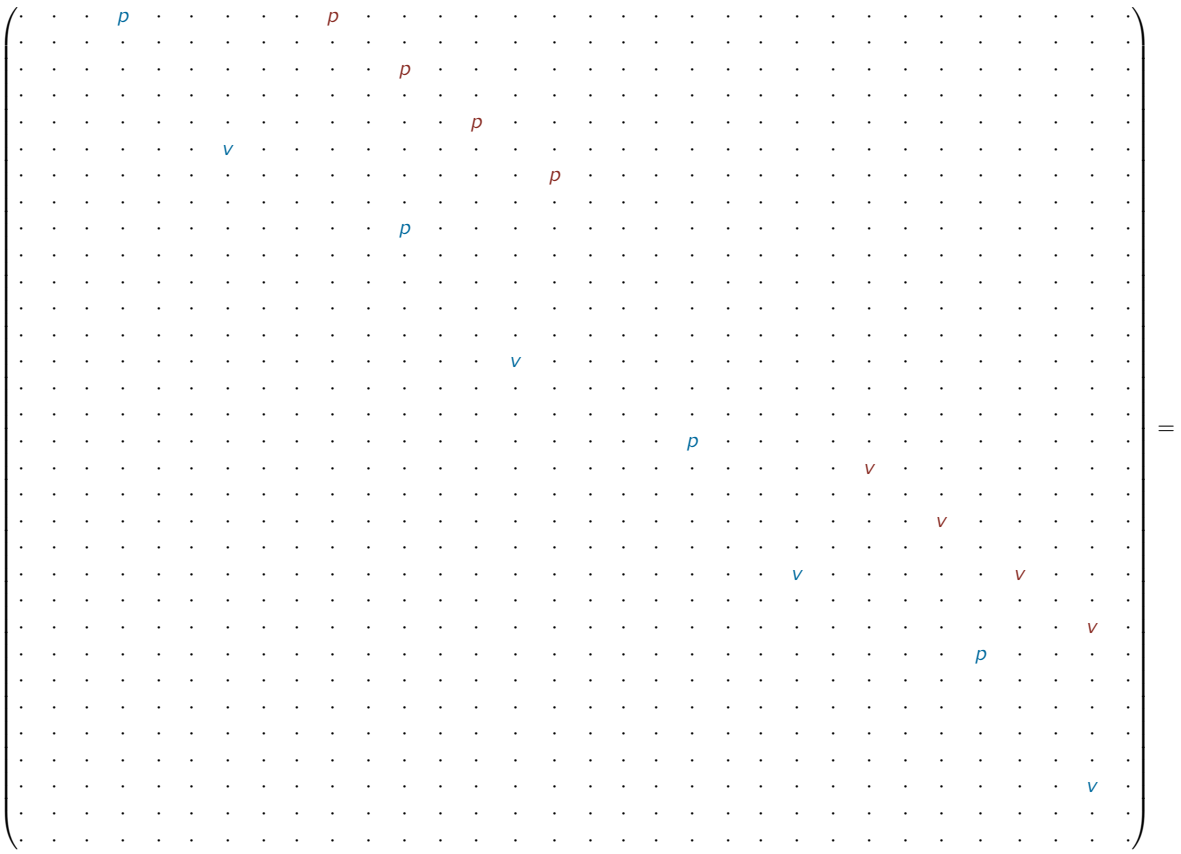
Figure 6.7: Graphical Representation of $(A \oplus B) \odot S$

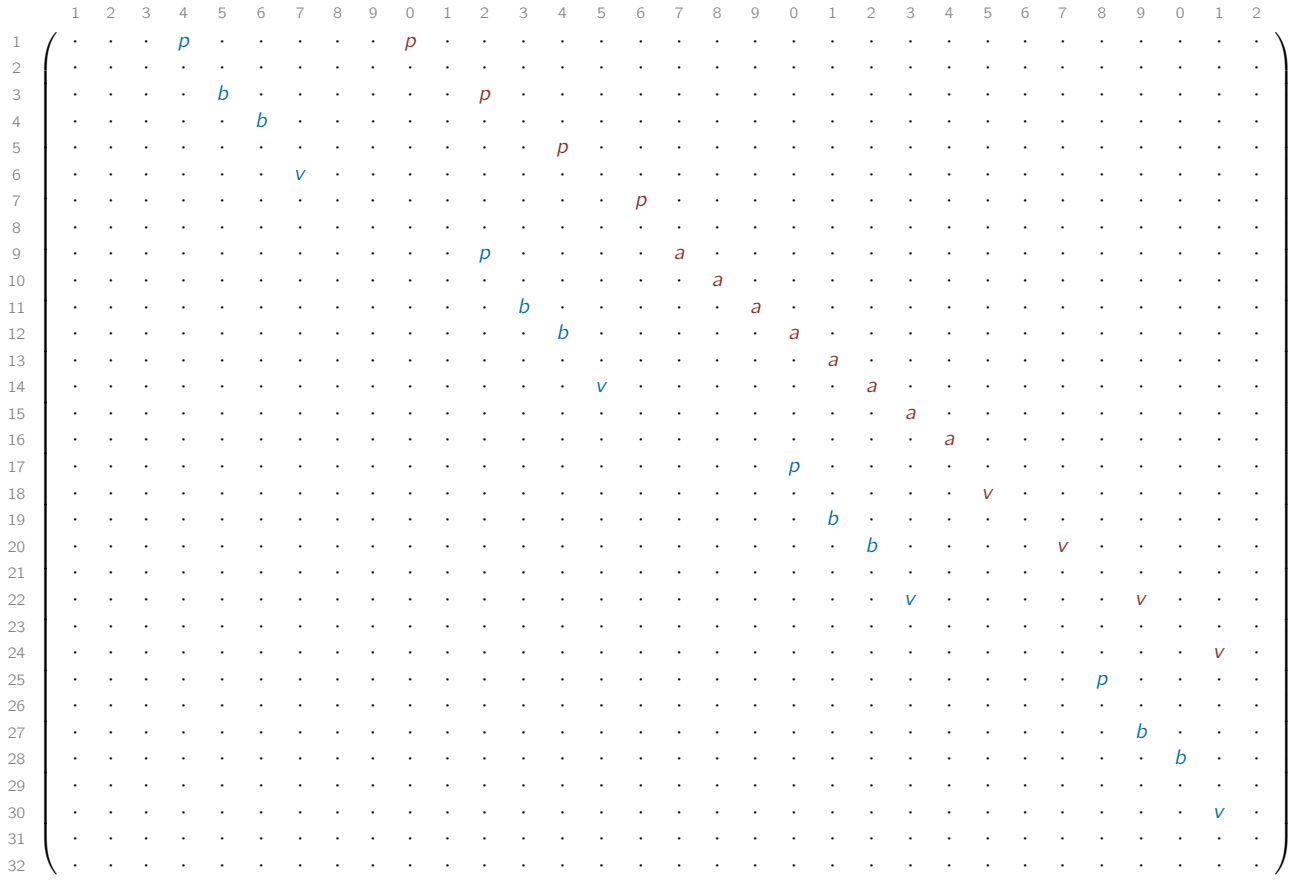


The next step is to calculate









In Figure 6.7 we depict the graphical representation of $(A \oplus B) \odot S$. Even without any knowledge on how semaphores work, one sees that either A gets a grip on the semaphore before B does or B gets it before A.

As an additional example consider two threads C and D with the CFGs presented in Figures 6.8 and 6.9. Both threads are using a semaphore for synchronization issues. The CFG of the semaphore clearly is still that of Figure 6.4. The matrices of the threads are

$$C = \begin{pmatrix} \cdot & a & \cdot & \cdot \\ \cdot & \cdot & p & \cdot \\ \cdot & \cdot & \cdot & b \\ \cdot & v & \cdot & \cdot \end{pmatrix}$$

and

$$D = \begin{pmatrix} \cdot & c & \cdot & \cdot \\ \cdot & \cdot & p & \cdot \\ \cdot & \cdot & \cdot & d \\ \cdot & v & \cdot & \cdot \end{pmatrix}.$$

First we have to compute $C \oplus D$.

$$C \oplus D = C \otimes I_4 + I_4 \otimes D =$$

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

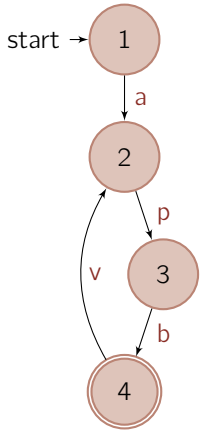


Figure 6.8: CFG of Thread *C*

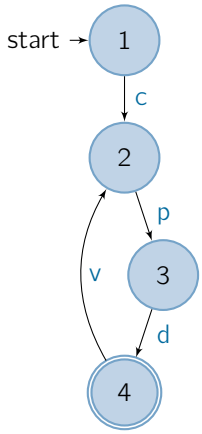
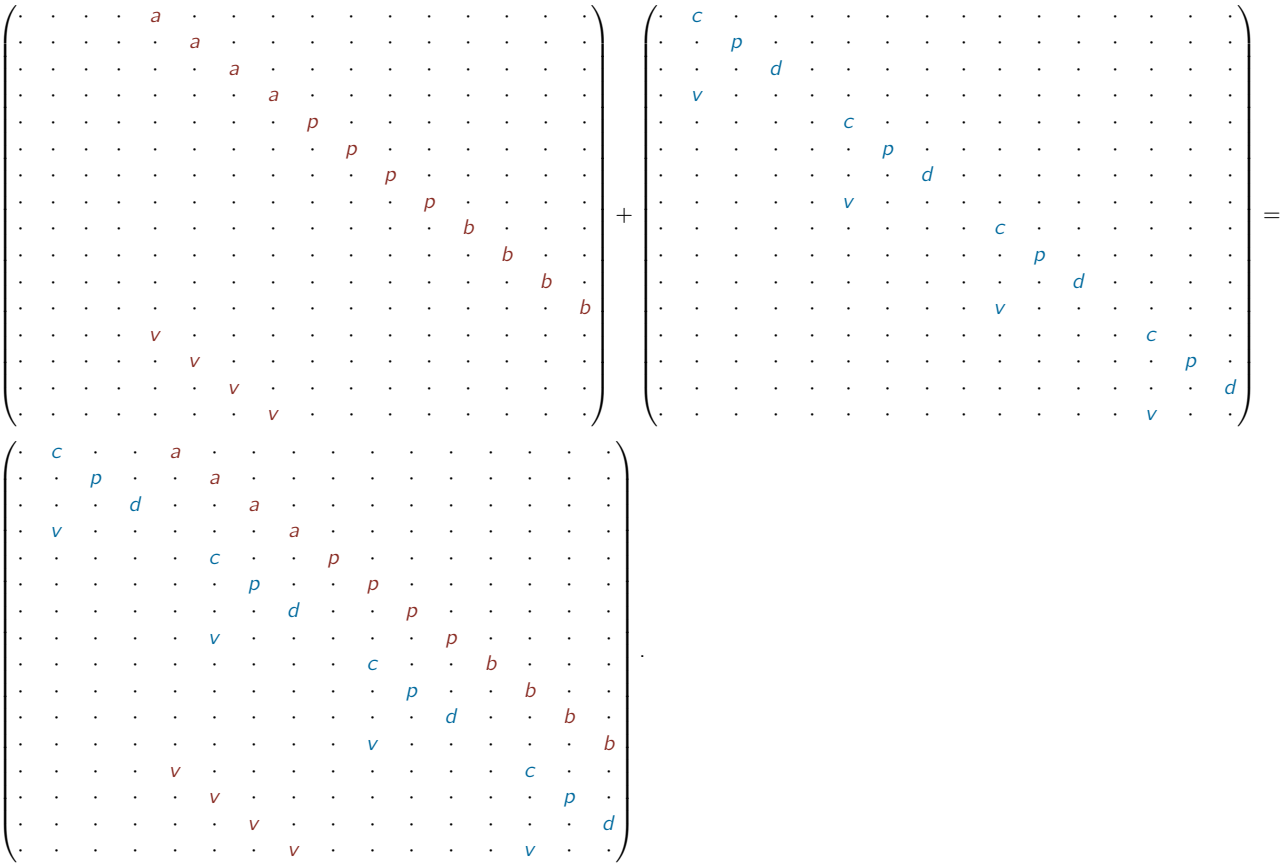
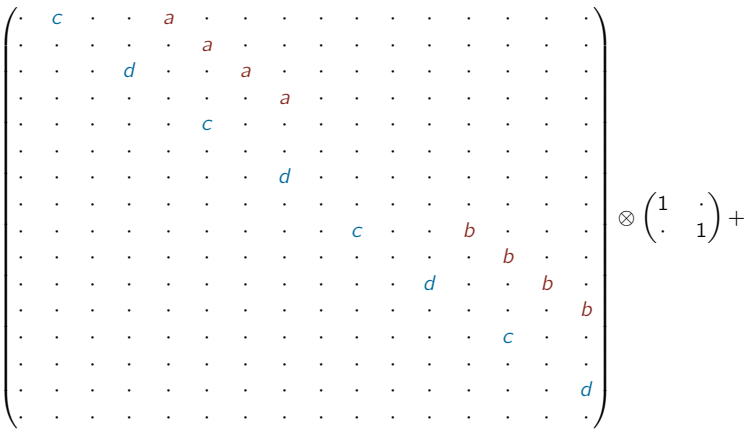


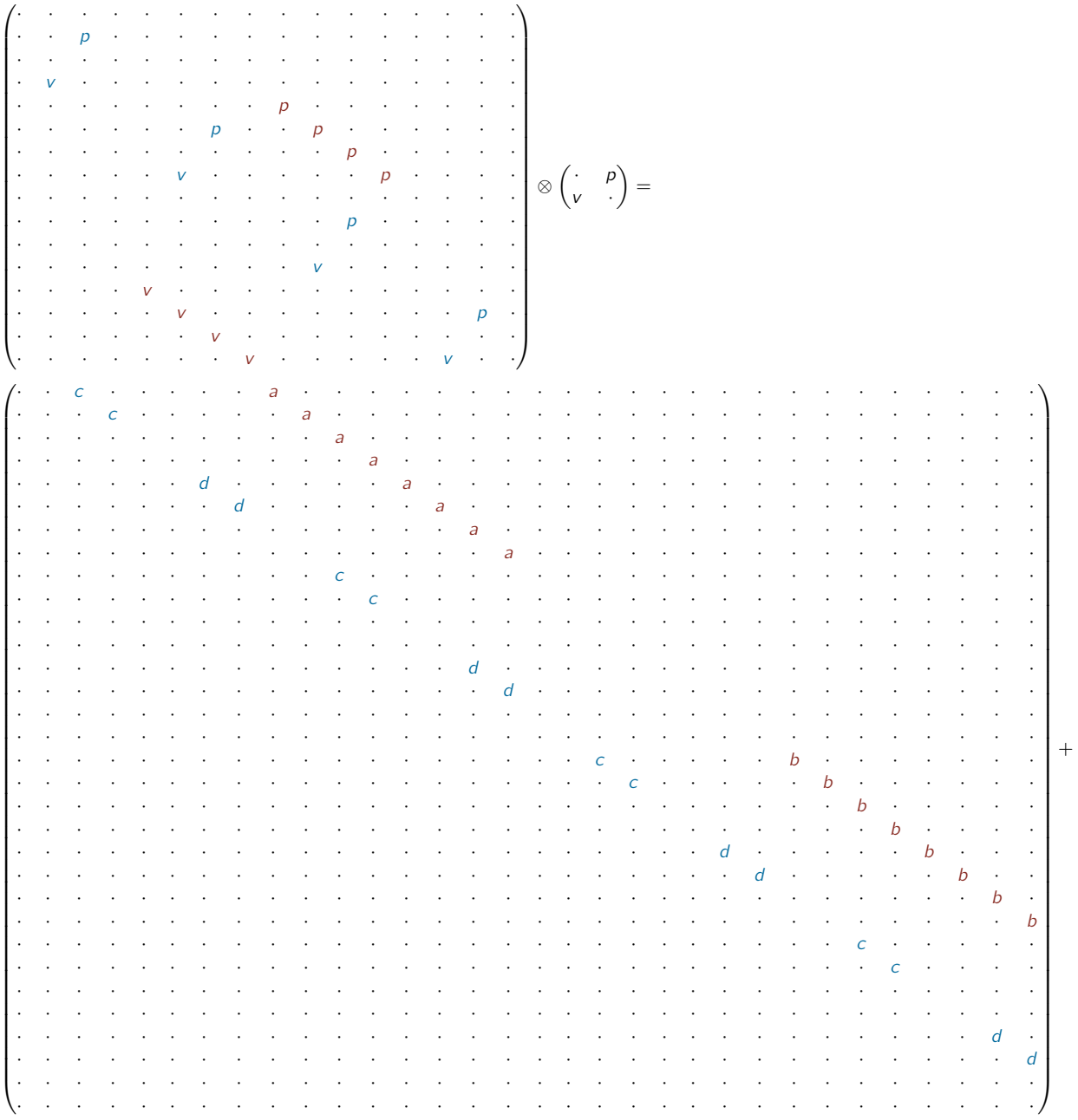
Figure 6.9: CFG of Thread *D*

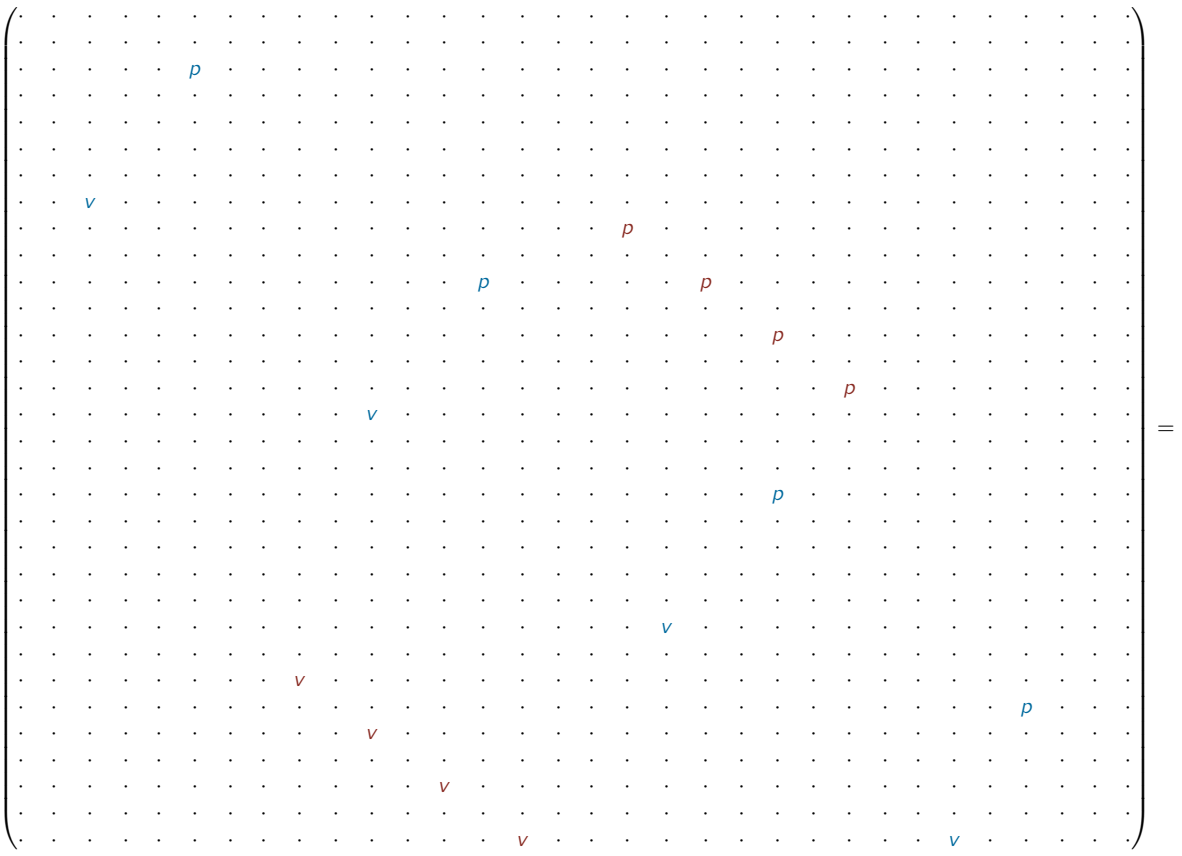


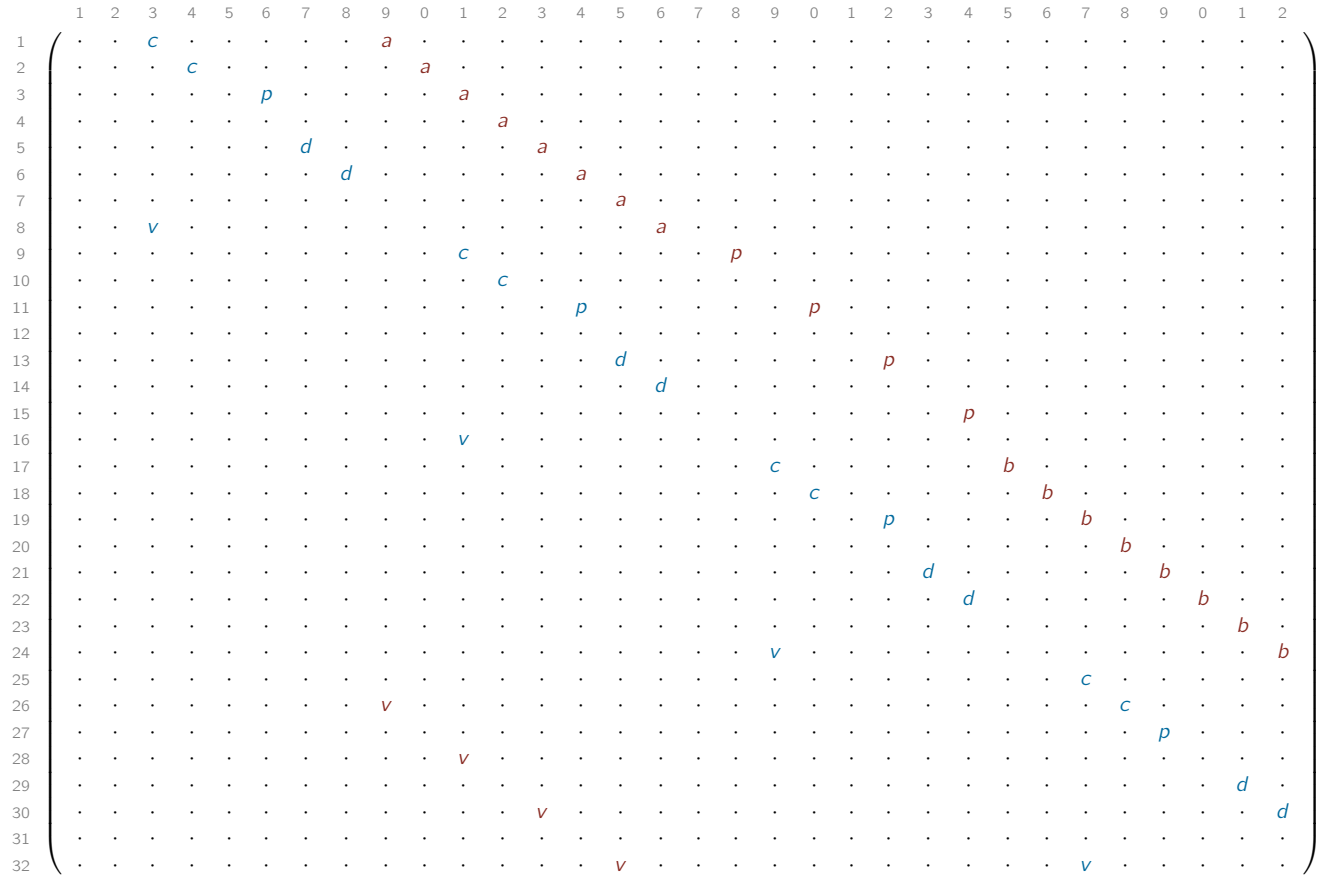
Next we have to apply the skip operation $\odot$. According to its definition we obtain

$(C \oplus D) \odot S =$









A graphical representation of $(C \oplus D) \odot S$ can be found in Figure 6.10. The reader is invited to check whether Figure 6.10 describes the behavior of the system as she expects it.

Thanks to Theorem 8 we are able to find a graph that models all interleavings of a finite number of threads, where the threads use a finite number of semaphores to synchronize each other. The next sections will be concerned with analyzes based on such a graph. Before we concentrate on these analyzes we have to name our graphs. We call the graph resulting from $\bigoplus_{i=1}^{n} T_i \odot \bigoplus_{j=1}^{k} S_j$ *Concurrent Program Graph* (CPG). The subgraph of a CPG defined by all parts of the CPG that can be reached from its initial node is called *Reachable CPG* (RCPG).

6.4 Race Conditions

Race conditions are problems that can occur in concurrent programs. Wikipedia has a nice definition of race conditions.

Definition 3. *A race condition or race hazard is a flaw in an electronic system or process whereby the output or result of the process is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first.*

Race conditions can occur in electronics systems, especially logic circuits, and in computer software, especially multithreaded or distributed programs.

In most cases, race conditions result from an erroneous use of synchronization primitives or from simply "forgetting" to synchronize an access to a variable shared between several threads.

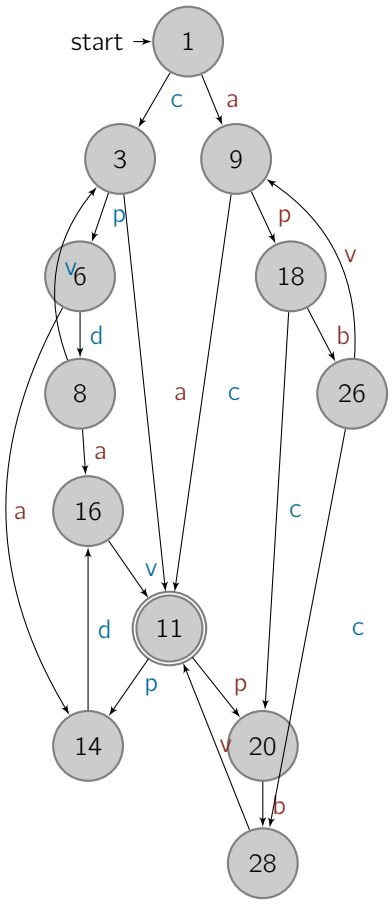


Figure 6.10: Graphical Representation of $(C \oplus D) \odot S$

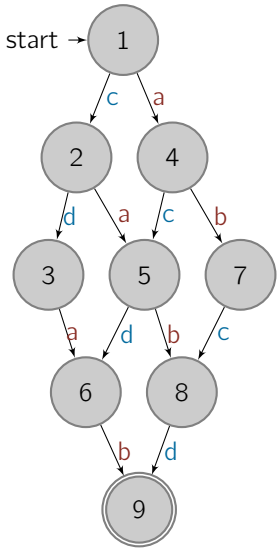


Figure 6.11: RCPG of a Simple Concurrent System

The RCPG of a simple concurrent system is shown in Figure 6.11. Assume that there exists a shared variable v initialized to 1. In addition, assume that the basic blocks a, b, c, and d contain the following updates to v:

$a \ \ldots v := v + 1$

$b \ \ldots v := v - 1$

$c \ \ldots v := 2 * v$

$d \ \ldots v := v/2$

Applying these updates to the RCPG in Figure 6.11 we obtain the graph shown in Figure 6.12. We see that at several nodes different values of v flow together. Clearly at each of these nodes a data race occurs.

We can setup a procedure based on RCPGs for finding data races.

- Find shared variables, i.e., variables read/written in several threads.
- Symbolically evaluate RCPG; variables may get "several" values.
- If the result has several values, a data race has been found.

We would like to add that finding shared variables can be done via a simple dataflow framework.

Herein we do not elaborate on

- how to symbolically evaluate loops,
- how to handle arrays, and
- how to handle pointers.

Some of these problems are studied in [BSB08].

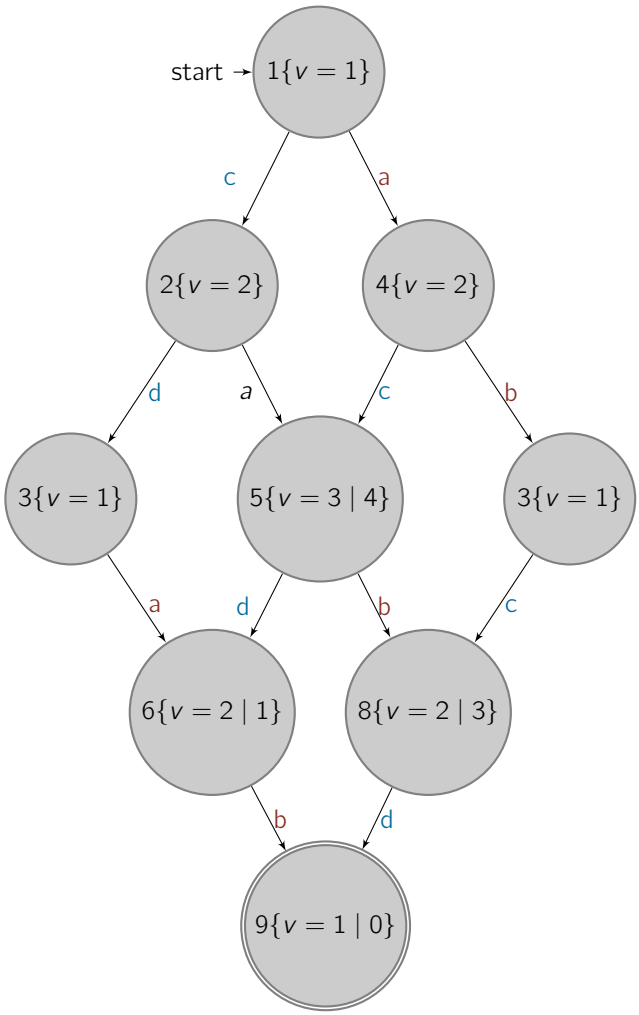


Figure 6.12: RCPG Equipped with Shared Variable Updates

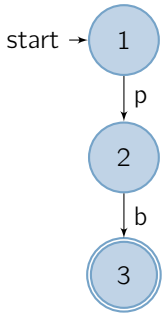


Figure 6.13: CFG of Thread B'

6.5 Misuses of Semaphores

In order to see how Kronecker analysis handles threads that do not use semaphores in a correct way, we study an example. Let Thread A be as given in Figure 6.5 and let Thread B of Figure 6.6 be modified. We assume that the programmer of Thread B forgot the semaphore v-operation. The resulting Thread B' is shown in Figure 6.13.

We obtain the matrices

$$A = \begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

and

$$B' = \begin{pmatrix} \cdot & p & \cdot \\ \cdot & \cdot & b \\ \cdot & \cdot & \cdot \end{pmatrix}.$$

We compute $A \oplus B'$

$$A \oplus B' = A \otimes I_3 + I_4 \otimes B' =$$

$$\begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \end{pmatrix} + \begin{pmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{pmatrix} \otimes \begin{pmatrix} \cdot & p & \cdot \\ \cdot & \cdot & b \\ \cdot & \cdot & \cdot \end{pmatrix} =$$

$$\begin{pmatrix} \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & p & \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 & a & \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 & v & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \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 & p & \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 & p & \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 & p & \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 & p & \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 \end{pmatrix} =$$

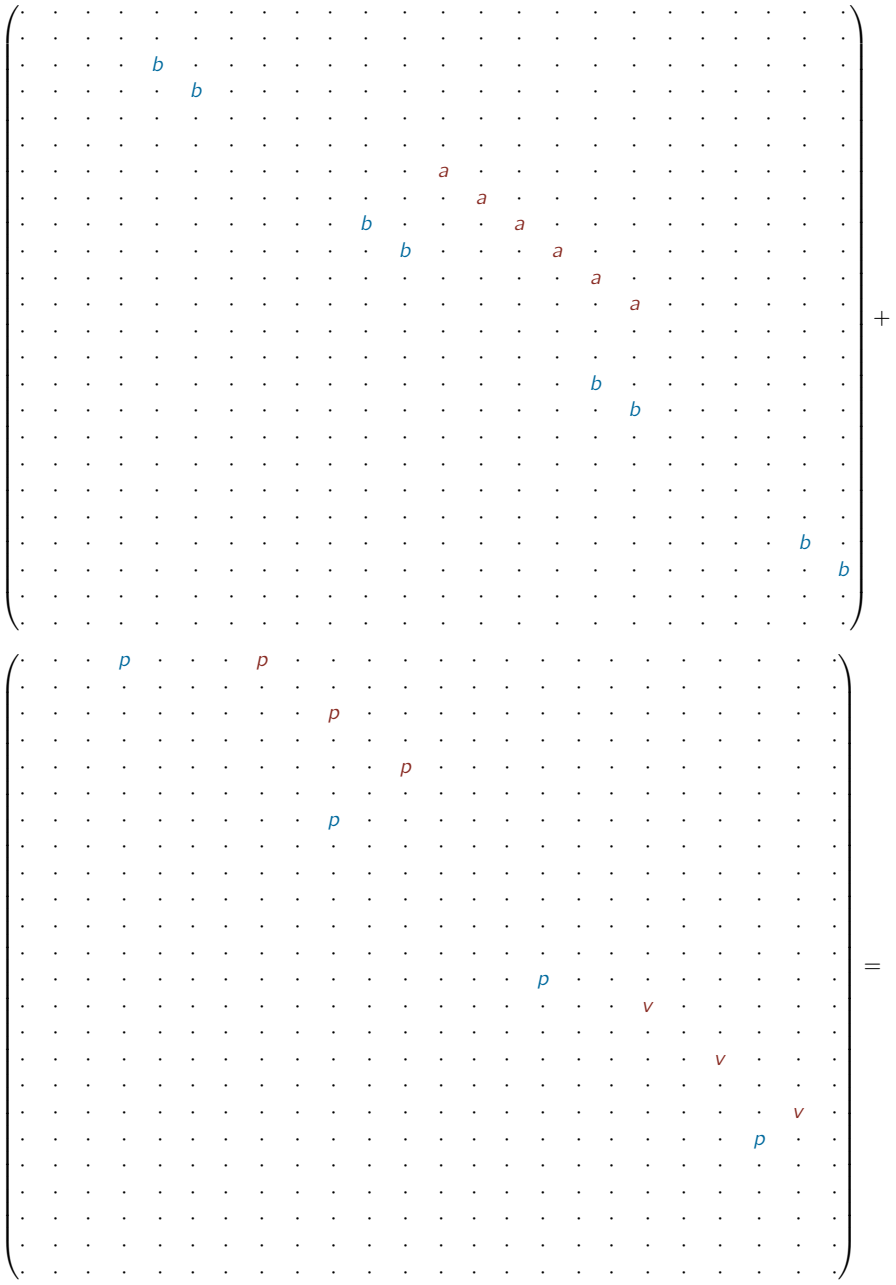
$$\begin{pmatrix}
\cdot & p & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & b & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot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

We continue by calculating

$$(A \oplus B') \odot S =$$

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

$$\begin{pmatrix}
\cdot & p & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & v & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix} \otimes \begin{pmatrix} \cdot & p \\ v & \cdot \end{pmatrix} =$$



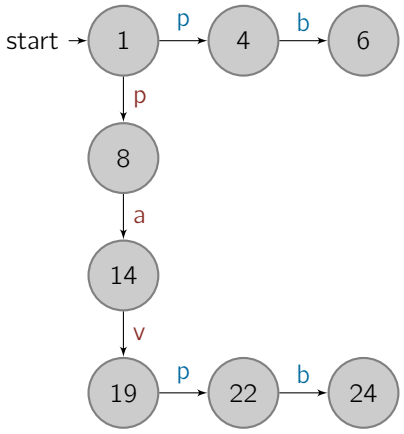


Figure 6.14: Graphical Representation of $(A \oplus B') \odot S$

$$
\begin{array}{c}
\begin{array}{cccccccccccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24
\end{array} \\
\left(\begin{array}{cccccccccccccccccccccccc}
1 & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
2 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
3 & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
4 & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
5 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
6 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
7 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
9 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
10 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
11 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
12 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
13 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
14 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot & \cdot & \cdot & \cdot & \cdot \\
15 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
16 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & v & \cdot & \cdot & \cdot \\
17 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
18 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot \\
19 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot \\
20 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
21 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot \\
22 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b \\
23 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
24 & \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{array}\right)
\end{array}
.
$$

A graphical representation of this matrix is depicted in Figure 6.14. Since the final node is supposed to be node 23 and this node is not present in Figure 6.14, we see that the (A, B')-program will not behave correctly. In more detail Figure 6.14 tells us that the (A, B')-program will be in a wrong "state" after b is executed. An experienced user of Kronecker analysis will be able to figure out why this happened, namely because a semaphore v-operation is missing.

In addition we would like to see what happens if a semaphore p-operation is forgotten. Let again Thread A be as given in Figure 6.5 and let Thread B'' be as shown in Figure 6.15.

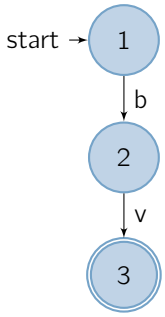


Figure 6.15: CFG of Thread B''

We obtain the matrices

$$A = \begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

and

$$B'' = \begin{pmatrix} \cdot & b & \cdot \\ \cdot & \cdot & v \\ \cdot & \cdot & \cdot \end{pmatrix}.$$

We compute $A \oplus B''$

$$A \oplus B'' = A \otimes I_3 + I_4 \otimes B'' =$$

$$\begin{pmatrix} \cdot & p & \cdot & \cdot \\ \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} 1 & \cdot & \cdot \\ \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \end{pmatrix} + \begin{pmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 \end{pmatrix} \otimes \begin{pmatrix} \cdot & b & \cdot \\ \cdot & \cdot & v \\ \cdot & \cdot & \cdot \end{pmatrix} =$$

$$\begin{pmatrix} \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & p & \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 & a & \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 & v & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \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 & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & v & \cdot & \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 & v & \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 & v & \cdot & \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 & v \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} =$$

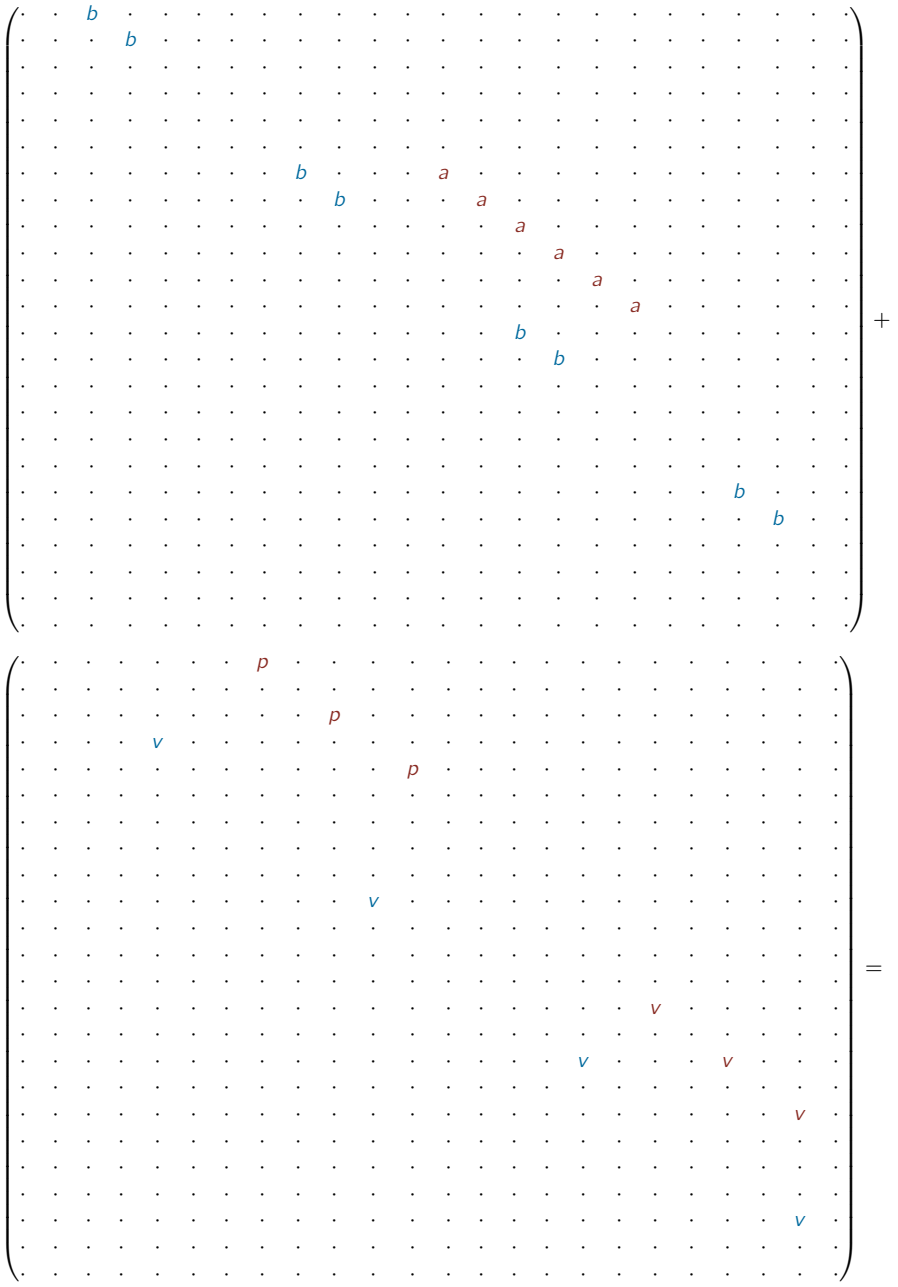
$$\begin{pmatrix}
\cdot & b & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & v & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot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

We continue by calculating the skip operation $\odot$

$$(A \oplus B'') \odot S =$$

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

$$\begin{pmatrix}
\cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & v & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot & v & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix} \otimes \begin{pmatrix} \cdot & p \\ v & \cdot \end{pmatrix} =$$



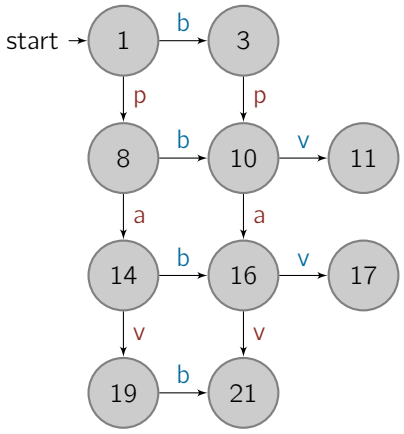


Figure 6.16: Graphical Representation of $(A \oplus B'') \odot S$

$$
\begin{array}{c}
\begin{array}{cccccccccccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24
\end{array} \\
\left(\begin{array}{cccccccccccccccccccccccc}
1 & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
2 & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
3 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
4 & \cdot & \cdot & \cdot & \cdot & v & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
5 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & p & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
6 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
7 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
9 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
10 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
11 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
12 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & a & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
13 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
14 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & v & \cdot & \cdot & \cdot & \cdot & \cdot \\
15 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
16 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot & \cdot & \cdot & v & \cdot & \cdot & \cdot \\
17 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
18 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot \\
19 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot & \cdot \\
20 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & b & \cdot & \cdot \\
21 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
22 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & v & \cdot \\
23 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
24 & \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{array}\right)
\end{array}
.
$$

A graphical representation of this matrix is depicted in Figure 6.16. Again the final node, node 23, is not present in Figure 6.16. Thus the (A, B'')-program is not correct and our programmer may now start debugging.

In general, we need some criteria to decide whether a concurrent system is "correct" or it is not. Until now the RCPGs of our incorrect example programs did not contain a final node. Can we be sure, that this always will be the case? No, sorry! If all threads contain at least one path such that the semaphores are used correctly within these paths, then our concurrent program will terminate, i.e., we will have a final node in the RCPG. So the presence of final nodes does not make a feasible criterion.

On the other hand, taking a closer look at Figures 6.14 and 6.16 we see that the RCPGs contain nodes from which

the final nodes cannot be reached[1]. We call such nodes *sink nodes*. It is clear that if an RCPG contains a sink node, the concurrent program is incorrect.

On the contrary, if a concurrent program is incorrect, can we be sure that the corresponding RCPG will contain at least one sink node? The answer is affirmative. If the program is incorrect, there is certainly a path not ending at one of the final nodes of the RCPG. Hence there has to be a sink node in the RCPG.

It remains to find an efficient way to spot sink nodes. One may classify sink nodes into two groups:

1. sink nodes with no successors and
2. sink nodes with at least one successor.

Now, sink nodes with no successors can be found easily: the corresponding line in the RCPG's matrix does contain only zeros, i.e., it is a *zero line*.

Sink nodes with successors cannot be handled that easily, but graph theorists and practitioners have tools to solve such problems. First of all we need some definitions.

Definition 4. *A directed graph is called* strongly connected *if there is a path in each direction between each pair of nodes of the graph. In a directed graph G that may not itself be strongly connected, a pair of nodes u and v are said to be strongly connected to each other if there is a path in each direction between them.*

The binary relation of being strongly connected is an equivalence relation, and the induced subgraphs of its equivalence classes are called strongly connected components *(SCCs). Equivalently, a strongly connected component of a directed graph G is a subgraph that is strongly connected, and is maximal with this property: no additional edges or nodes from G can be included in the subgraph without breaking its property of being strongly connected. The collection of strongly connected components forms a partition of the set of nodes of G.*

If each SCC is contracted to a single node, the resulting graph is a directed acyclic graph, the condensation *of G.*

We call an SCC that contains at least one final node *final SCC*. Now, we consider the condensation of an RCPG, called cRCPG, and get the following theorem.

Theorem 9. *A cRCPG contains a node from which no final node can be reached if and only if the concurrent program is incorrect.*

Finally we mention that SCCs can be found in linear time, e.g. by using Tarjan's algorithm [Tar72].

So far we considered concurrent programs incorrect only when semaphore operations were employed in a wrong way. But there are additional problems with concurrent programs. Although a concurrent program employs semaphore operations correctly, it still may get stuck. Such deficiencies are called *deadlocks*. We will study them in the next section.

6.6 Deadlocks

Design and implementation of concurrent systems are extremely difficult tasks. There are several reasons for this. One is data races. Another one is that the programmer has to use the means for synchronization in a correct way. In the previous sections we have developed Kronecker based methods for finding data races and for identifying incorrect uses of semaphores. However concurrent systems generate characteristic problems that cannot be attributed to incorrect use of synchronization means. Deadlocks belong to these problems. Deadlock is also known as deadly embrace. A system must meet four conditions for a deadlock to occur [Sta01]:

1. Mutual exclusion: a resource that cannot be used by more than one process at a time.

[1] Actually both RCPGs do contain only such nodes.

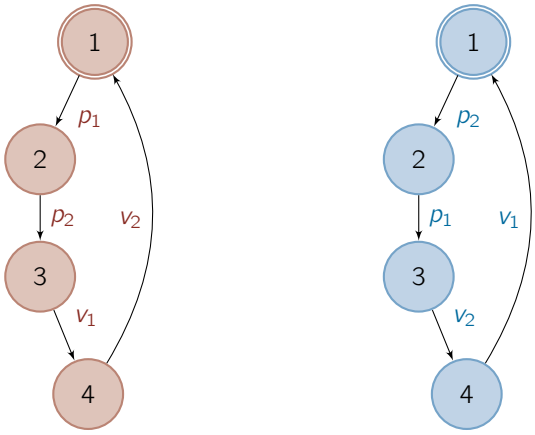


Figure 6.17: Graphical Representation of Two Dining Philosophers

2. Hold and Wait: processes already holding resources may request new resources held by other processes.

3. No preemption: no resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process.

4. Circular wait: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.

In order to prohibit deadlocks it is enough to ensure that one of these conditions cannot occur [San11]. However there are systems for which this cannot be achieved. For such systems a check for being free of deadlocks is requiredIn the following we study a prototypical example for deadlocks, the famous *dining philosophers problem*.

Consider n philosophers sitting around a table. The philosophers normally are contemplating on questions like "How many deadlocks can dance on the head of a pin?". But when a philosopher gets hungry, he grabs two forks and starts eating spaghetti. Since someone always is delivering spaghetti, there is no problem with food supply. The problem occurs because of the forks. Each philosopher has to grab one on his left and one on his right side, but only one fork has been placed between two philosophers. A philosopher first grabs the fork on the left, then that on the right. As soon as a philosopher has grabbed two forks, he starts eating. When he is finished, he puts down the forks so they can be used by others[2].

Now, if all n philosophers get hungry at the same time, all of them will grab the fork on their left. After that no one can grab a fork on his right, because that is already held by his neighbor. So the system deadlocks and our poor philosophers starve to death, leaving us behind with many unanswered questions on deadlocks and pins.

The smallest dining philosophers examples consists of two semaphores modelling the forks and two threads modelling the philosophers. A graphical representation of the philosophers can be found in Figure 6.17. Since the size of one semaphore matrix is 2 and that of one philosopher is 4, we obtain a size of the overall system of $4 \cdot 4 \cdot 2 \cdot 2 = 64$, a little bit to big to be presented herein in full detail. However, the calculations are straight-forward and we obtain the graphical representation shown in Figure 6.18.

Clearly, again there is a node, node 24, from that the final node, node 1, cannot be reached. Thus, the deadlock is found. A closer inspection reveals that the deadlock occurs if philosopher 1 grabs fork 1 and before he can grab

[2]No comments on dish washing and hygiene, please.

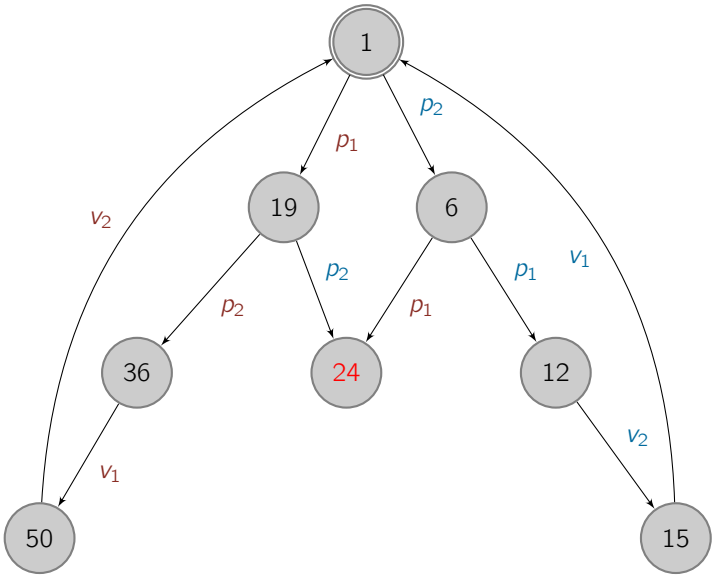


Figure 6.18: Graphical Representation of the Two Dining Philosophers system

fork 2, philosopher 2 grabs it. There is a second way for the deadlock to occur: philosopher 2 grabs fork 2 and before he can grab fork 1, philosopher 1 grabs it. These cases coincide with the scenario we have foreseen above.

In general, the ideas we have devised in the previous section apply also to Kronecker based deadlock analysis. Hence we state the following theorem.

Theorem 10. *A cRCPG contains a node from which no final node can be reached if and only if the concurrent program is incorrect.*

In this and in the previous section we have presented a Kronecker based method for detecting deadlocks and for finding misuses of semaphores. Both are based on so-called reachability analysis. However, semaphores are not the only means for synchronizing threads. Higher-level constructs are monitors. The next chapter will concentrate on how to model and analyze concurrent systems which use monitors for synchronization.

Exercises

1. Find a CFG representation for a *counting semaphore* of size n.

Chapter 7

Lazy Implementation

One of the major problems of Kronecker algebra is the size of the matrices. Even small examples produce big matrices. Most of them are sparse, i.e., they contain only few non-zero elements. For the railway and software domain usually only $O(s)$ elements of a matrix of size s are non-zero. In most cases the number of non-zero entries is even smaller. Even if standard memory saving methods are applied for spares matrices, memory demand is still too high.

For this reason, our implementation is based on *lazy evaluation* [HM76]. Basically this means that the matrix expressions are not evaluated at all. The only thing to be stored are the matrices modelling the threads or train routes and the matrices modelling the synchronization constructs (semaphores, monitors, track sections, ...). In addition, for these we can employ standard memory saving methods for spares matrices. Details can be found in the following sections.

7.1 Expression Trees

Instead of evaluating a matrix expression like $A \otimes S$ we simply store an *expression tree* equivalent to the expression. Simple examples are presented in Figures 7.1 and 7.2. Note that we only need to handle the matrix operations "+" and "$\otimes$" because "$\odot$" and "$\oplus$" are defined via "+" and "$\otimes$".

7.2 Lazy Evaluation of Kronecker Expressions

In order to build a graphical representation of the result of such a matrix expression, we need to have two operations:

Successor-Operation to find the successors of node i, i.e., the non-zero entries of line i, and

Access-Operation to return the matrix entry (i, j) (line i and column j).

Let's start with "+". Assume that we have an expression $A+B$ where A and B are matrices. To find the non-zero entries of line i in $A+B$, we need to find the non-zero entries of line i in matrix A and in matrix B. The set of non-zero entries of line i in $A+B$ then simply is the union of the sets of the non-zero entries of line i in matrix A and in matrix B.

To calculate the entry (i, j) of $A+B$, we only need to find the entry (i, j) in A and the entry (i, j) in B. If we sum up the latter two, we obtain our result.

So this was easy, but how can we handle the Kronecker product $\otimes$? This is a little bit more complex, but can be achieved too.

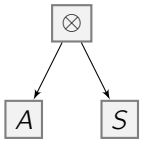


Figure 7.1: Expression Tree for $A \otimes S$

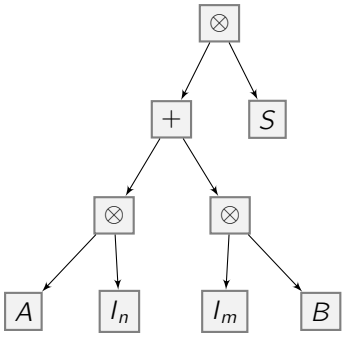


Figure 7.2: Expression Tree for $(A \oplus B) \otimes S = (A \otimes I + I \otimes B) \otimes S$

Assume we want to find the successors of node i in the matrix $A \otimes B$. In addition, let the size of matrix A be denoted by m and that of B by n. The block matrix has the following structure

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

The first question to answer is: Which of the $a_{s,t}$ do we have at line i of $A \otimes B$? But, since the matrix is built up by blocks of the same size, namely n, we can give the answer. We simply have $s = ((i-1) \div n) + 1$, where $\div$ denotes integer division.

The second question is: Which $b_{u,v}$ to we have at line i of $A \otimes B$? Again, the answer is simple. We have $u = (i \bmod n)$ if $u > 0$[1]. If u turns out to be 0 by the above computation, we set $u = n$.

Thus, we have found which lines of A and B to investigate in order to find the successors of node i. In matrix A we have to examine line s and in matrix B line u.

Let the set of successors of i be denoted by $\mathcal{S}_i$ and let $\mathcal{S}_i$ be initialized to $\emptyset$.

Now, in a loop we take a closer look at $a_{s,t}$ for $1 \leq t \leq m$:

If $a_{s,t} = 0$, then $a_{s,t} \cdot b_{u,v} = 0$. So this will not contribute to the set of successors.

If $a_{s,t} \neq 0$, then in a loop we check $b_{u,v}$ for $1 \leq v \leq n$: If $b_{u,v} \neq 0$, v is added to the set of successors, i.e., $\mathcal{S}_i := \mathcal{S}_i \cup \{v\}$.

In the end, $\mathcal{S}_i$ contains the set of successors of node i.

By a similar line of thought, we are able to calculate the entry (i, j) of $A \otimes B$.

[1] $i \bmod n$ denotes the remainder we get if we divide i by n.

Summing up, we have learned how to retrieve all necessary information from simple expression trees like $A + B$ or $A \otimes B$ to build up a graphical representation of graphs. But how can we handle expression trees like that in Figure 7.2 or even more complex ones?

The answer is recursion. We handle successor and access operations at the root node by employing successor and access operations to the left and right subtree of the root node. The root node of the left and right subtree then can be handled the same way, and so forth. When finally we arrive at a leaf node, we can find successors and entries as usual, because standard matrices are situated at the leaves of an expression tree.[2]

This *lazy implementation* of Kronecker matrix expressions turned out to be very time and memory efficient. Note, however, that integer node ids are still necessary for the whole process of deriving graphical representations. Our practical experiments soon showed that 64 bits integers are too small to work with medium sized problems. So we had to implement operations for big integer node ids that do not fit into 64 bits.

Ideas how to speed up an implementation even more are given in the next section.

7.3 Parallelizing

Matrix operations are easy to parallelize for multi-processor computers. Actually there is a plethora of literature on that issue. So it does not come as a surprise that Kronecker operations can also be parallelized easily.

Our implementation of generating a graphical representation of a Kronecker matrix expression is based on two work-lists. L_1 holding the nodes that already have been processed, is initialized to $\emptyset$. The other, L_2, holding nodes that still need to be processed is set to the initial node at the beginning.

The algorithm fetches nodes from L_2 as long as $L_2 \neq \emptyset$ thereby removing these nodes from L_2 and hands them one at the time to a *worker* thread. The number of worker threads is limited by the number of CPUs (cores, ...) that are assigned to the algorithm.

Each worker calculates the successors of the node it got and inserts them into L_2 if they are not already present in L_1. By accessing entries of the matrix expression in the way we have presented in the previous section, the worker finds all edge labels for all successor nodes. Finally the worker inserts the node it just has processed, into L_1.

In addition, by making sure that a certain node is always processed by the same thread, work-list L_1 can be replaced with work lists being local to the threads. This reduces the synchronization effort and speeds up the implementation significantly. By choosing a lock-free implementation for L_2 we have further enhanced the performance of our implementation.

Finally, taking into account the specific hardware architecture of GPGPUs, we are now able to exploit the enormous number of cores of graphic cards for our purpose of Kronecker algebra computations.

Exercises

1. What are the formulas to calculate the entry (i, j) of $A \otimes B$?

[2] Actually the matrices will be represented by memory saving data structures for sparse matrices, but we need not bother with that at the moment.

Chapter 8

Historical Remarks

After Georg Zehfuss' pioneering work [Zeh58] in 1858, Leopold Kronecker used the direct matrix product frequently in his lectures. So his students ascribed it to him and the name *Kronecker product* was coined [Knu11]. In 1894 Adolf Hurwitz used *Kronecker sum* in a paper [Hur94]. This is the reason why Kronecker sum is sometimes named after him. In 1985 Brigitte Plateau employed Kronecker algebra in the context of stochastic automata [Pla85].

All of the above were applying Kronecker algebra to matrices with entries from a numerical domain. The first to study Kronecker sum in the context of automata was Gerhard Küster [Küs86, Küs91] in 1986. He used the name Hurwitz product and studied it within the Kuich-Salomaa notation [KS86] that based automata theory on purely algebraic foundations. Küster provided the first formal proof that Kronecker sum generates all interleavings of concurrently executing automata. In 2002, Peter Buchholz and Peter Kemper [BK02] applied Kronecker algebra for efficiently determining reachability sets of automata. They used matrices with Boolean entries.

The first paper applying Kronecker algebra, i.e., Kronecker product and Kronecker sum, to automata within the Kuich-Salomaa notation seems to be [MB11]. Here both operations are defined over a semiring. Kronecker sum is used for modeling concurrency and Kronecker product is used for synchronizing purposes. However, Kronecker product for synchronization purposes was already employed in [Pla85] and in [BK02].

Since then many different applications of Kronecker algebra have been studied.

- How to analyze concurrent programs that employ monitor-like synchronization has been described in [BB14].
- Analyzing railway systems has been done in several papers e.g. [MBS12, VBS12, VBS13, VBS14].
- How WCET analysis of concurrent programs can be done via Kronecker algebra has been shown in [MB14].

PS. It is very likely that the author has missed interesting papers on Kronecker product or Kronecker sum that may be considered predecessors of the current work on Kronecker algebra and its applications. The author may also have completely misunderstood the contents of the papers cited above. He appreciates any hint from the readers.

Figure 8.1: Leopold Kronecker
⋆ 7.12.1823 Legnica, Poland;
† 29.12.1891 Berlin, Germany

Figure 8.2: Adolf Hurwitz
⋆ 26.3.1859 Hildesheim, Germany;
† 18.11.1919 Zürich, Switzerland

Bibliography

[BB14] Johann Blieberger and Bernd Burgstaller, *Kronecker Algebra for Static Analysis of Ada Programs with Protected Objects*, Ada-Europe'2014 International Conference on Reliable Software Technologies, LNCS 8454 (Paris, France), June 2014, pp. 27–42.

[BK02] Peter Buchholz and Peter Kemper, *Efficient Computation and Representation of Large Reachability Sets for Composed Automata*, Discrete Event Dyn. Systems **12** (2002), no. 3, 265–286.

[BSB08] Bernd Burgstaller, Bernhard Scholz, and Johann Blieberger, *Symbolic Analysis: An Algebra-Based Approach*, VDM Verlag, Saarbrücken, 2008.

[HIK11] Richard Hammack, Wilfried Imrich, and Sandi Klavžar, *Handbook of product graphs*, 2nd ed., CRC Press, Boca Raton, London, New York, 2011.

[HM76] Peter Henderson and James H. Morris, Jr., *A Lazy Evaluator*, 3rd ACM Symposium on Principles of Programming Languages, POPL '76, January 1976, pp. 95–103.

[HT66] Frank Harary and Charles A. Trauth, Jr., *Connectedness of products of two directed graphs*, SIAM Journal on Applied Mathematics **14** (1966), no. 2, 250–254.

[Hur94] Adolf Hurwitz, *Zur Invariantentheorie*, Math. Annalen **45** (1894), 381–404.

[Knu11] Donald E. Knuth, *Combinatorial algorithms*, The Art of Computer Programming, vol. 4A, Addison-Wesley, 2011.

[KS86] Werner Kuich and Arto Salomaa, *Semirings, Automata, Languages*, Springer, 1986.

[Küs86] Gerhard Küster, *Das Hadamardprodukt abstrakter Familien von Potenzreihen*, Ph.D. thesis, TU Vienna, 1986.

[Küs91] ———, *On the Hurwitz Product of Formal Power Series and Automata*, Theor. Comput. Sci. **83** (1991), no. 2, 261–273.

[MB11] Robert Mittermayr and Johann Blieberger, *Shared Memory Concurrent System Verification using Kronecker Algebra*, Tech. Report 183/1-155, Automation Systems Group, TU Vienna, http://arxiv.org/abs/1109.5522, Sept. 2011.

[MB14] ———, *Worst-Case Execution Time Analysis of Concurrent Programs*, (submitted).

[MBS12] Robert Mittermayr, Johann Blieberger, and Andreas Schöbel, *Kronecker Algebra based Deadlock Analysis for Railway Systems*, PROMET-TRAFFIC & TRANSPORTATION (2012), 359–369.

[McA63] M. H. McAndrew, *On the product of directed graphs*, Proceedings of The American Mathematical Society **14** (1963), no. 4, 600–606.

[Pla85] Brigitte Plateau, *On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms*, ACM SIGMETRICS, vol. 13, 1985, pp. 147–154.

[San11] Bo I. Sandén, *Design of multithreaded software*, John Wiley & Sons, Hoboken, New Jersey, 2011.

[Sta01] William Stallings, *Operating systems*, 4th ed., Prentice-Hall, Upper Saddle River, New Jersey, 2001.

[Tar72] Robert Endre Tarjan, *Depth-First Search and Linear Graph Algorithms*, SIAM J. Comput. **1** (1972), no. 2, 146–160.

[VBS12] Mark Volcic, Johann Blieberger, and Andreas Schöbel, *Kronecker Algebra based Travel Time Analysis for Railway Systems*, FORMS/FORMAT 2012 – 9th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (Braunschweig, Germany), December 2012, pp. 273–281.

[VBS13] ———, *Kronecker algebra and its broad applications in railway systems*, EURO-ŽEL 2013: Recent Challenges for European Railways (Žilina, Slovak Republic), June 2013, pp. 275–282.

[VBS14] ———, *Optimisation of railway operation by application of Kronecker algebra*, CETRA 2014 (Split, Croatia), April 2014, pp. 37–42.

[Wei62] Paul M. Weichsel, *The Kronecker product of graphs*, Proceedings of the American Mathematical Society **13** (1962), no. 1, 47–52.

[Zeh58] Johann Georg Zehfuss, *Ueber eine gewisse Determinante*, Zeitschrift für Mathematik und Physik **3** (1858), 298–301.