

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Performance analysis of parallel blocksorters

by
Martijn Wolfs

Supervisor:

drs. R.H. Mak (TU/e)

Eindhoven, March 2007

Abstract

A method to analyze the performance of various designs of stream processing systems that perform block computations is presented in this thesis. Various performance metrics are given: capacity, i/o-distance, throughput, cycle time, latency, occupancy and elasticity. The thesis' focus is primarily on the latter metric, elasticity, which gives an indication of how well a design can cope with varying throughput in its environment. With the exception of capacity and i/o-distance, the performance metrics given depend on schedules, mappings of the events of a system to time slots.

Providing a purely theoretical analysis of the performance of a design can be very time-consuming. Such an analysis can provide bounds for the values of certain performance metrics, but whether a design can achieve values that are equal to these bounds is sometimes not clear. A different type of analysis is given in this thesis. One advantage of this analysis is that some of its steps can be automated. The analysis makes use of so called extremal schedules that achieve minimal or maximal average latency for a fixed average throughput. These schedules are computed by solving specially constructed integer linear programming problems. A compiler that transforms program texts describing stream processing systems into these linear programming problems is built for this purpose. This thesis thoroughly describes the techniques used to go from program texts to integer linear programming problems.

The compiler is used to calculate schedules for various instances of systems from the class of bubble block sorters. The elasticity is computed using these schedules and the values are compared to bounds obtained from a theoretical analysis of these bubble block sorter systems. The outcome of these experiments shows rich behavior for the various bubble block sorter systems and a thorough explanation is provided for various aspects of the results obtained from the experiments.

Preface

This document is my master's thesis, describing my final work for Computer Science And Engineering at the Eindhoven University of Technology. The research was done within the System Architecture and Networking group of the Department of Mathematics and Computer Science under the supervision of Drs. R.H. Mak.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Stream processing systems | 1 |
| 1.1.1 | Construction method | 1 |
| 1.1.2 | Communication constraints | 2 |
| 1.1.3 | Block computations | 2 |
| 1.2 | Problem description | 3 |
| 1.2.1 | Performance analysis | 4 |
| 1.2.2 | Project goal | 6 |
| 1.3 | Document overview | 7 |
| 2 | System structure | 9 |
| 2.1 | Basic building blocks | 9 |
| 2.1.1 | Component attributes | 12 |
| 2.2 | System composition | 13 |
| 2.2.1 | System attributes | 15 |
| 3 | System behavior | 17 |
| 3.1 | Events | 17 |
| 3.2 | Precedence graphs | 18 |
| 3.3 | Derived precedence graphs | 20 |
| 3.4 | Event scheduling | 21 |
| 3.4.1 | Component schedules | 25 |
| 3.4.2 | System schedules | 29 |
| 4 | System performance | 31 |
| 4.1 | Performance metrics | 31 |
| 4.1.1 | Capacity | 31 |
| 4.1.2 | I/o-distance | 31 |
| 4.1.3 | Cycle time and throughput | 31 |
| 4.1.4 | Latency and occupancy | 33 |
| 4.1.5 | Elasticity | 35 |
| 5 | Integer linear programming problems | 37 |
| 5.1 | ILP problems of individual components | 38 |
| 5.2 | Constructing programming problems of combined systems | 43 |
| 5.3 | ILP solver | 45 |

| | | |
|----------|---|-----------|
| 6 | Tool | 47 |
| 6.1 | Parser | 47 |
| 6.1.1 | Checking the syntax | 48 |
| 6.1.2 | Verifying context conditions | 48 |
| 6.1.3 | Constructing the attributed parse tree | 48 |
| 6.2 | Generating the ILP from a parse tree | 48 |
| 7 | Results | 51 |
| 7.1 | Bubble block sorter systems | 51 |
| 7.1.1 | Performance of bubble block sorter systems | 52 |
| 7.2 | General approach | 54 |
| 7.3 | Calculating actual values for performance metrics | 55 |
| 7.3.1 | Experiment description | 56 |
| 7.3.2 | Experiment results | 56 |
| 7.4 | Comparing performance metrics with the bounds | 57 |
| 7.4.1 | Experiment description | 57 |
| 7.4.2 | Experiment results | 57 |
| 8 | Conclusion | 65 |
| 8.1 | Future Research | 67 |
| A | Grammar | 71 |
| A.1 | Attribute domains | 71 |
| A.2 | Nonterminals (with attribute domains) | 71 |
| A.3 | Terminals | 72 |
| A.4 | Start symbol | 73 |
| A.5 | Production rules and context conditions | 73 |
| B | Tokens | 81 |

Chapter 1

Introduction

This chapter starts of with some background information about the research done for my master's thesis, followed by the problem description. The final section of this chapter gives a detailed overview of the structure of the rest of this thesis.

1.1 Stream processing systems

The systems under consideration in this thesis belong to the class of systems called stream processing systems (**SPSs**) [10]. **SPSs** are systems that convert one or more streams of input data to one or more streams of output data. This is done in a parallel manner; several parts of the system can be active at the same time. Stream processing systems are often implemented in hardware and typical applications are found in digital signal processing, image processing and filtering. Another example of stream processing systems are UNIX pipes [11], a stream of output from one application is the stream of input for another application.

To delimit the domain of systems researched in this thesis, we will only be looking at stream processing systems constructed in a special manner and that satisfy specific constraints on their communications.

1.1.1 Construction method

The stream processing systems that are investigated in this thesis are constructed in a special, LEGO[®] like manner, that is, by taking various small building blocks, called components, and combining them to form systems that meet some functional requirements. These components are very simple with respect to the functionality they provide.

Some examples of these components are a one-place buffer, which reads a value from its input port and writes that same value on its output port, a split component that divides an input stream into two output streams or a merge component that combines two input streams into one output stream. Note that these components do not change the values of the input stream, they merely pass them on.

The comparator (also called compare and swap component) is a component with a little more functionality. It has two input and two output ports and reads one value from each input port and writes the maximum of the values to one and the minimum of the values to the other output port.

With only a limited number of these different components, large and complex systems can be constructed that perform a variety of different tasks.

1.1.2 Communication constraints

A stream processing system that is constructed by combining components can do its work because the components communicate. Communications between the components take place on uni-directional channels. These channels connect the output port of one component to the input port of another component. Each channel is connected to two ports that either belong to the same component or to two distinct components. To limit the classes of systems that are discussed, there are limitations put on the communication structure and the communication behavior of the components.

Communications between components of the system take place in a manner similar of that in Communicating Sequential Processes [2], i.e., we require the communications to be synchronous, which means that a communication only takes place if both parties involved in the communication are ready to communicate. This means that for synchronous communications, unlike asynchronous communications, a channel has no storage capacity.

Furthermore, we require that the communication behavior of the system is data-independent, by which we mean that it does not depend on the values that are communicated.

1.1.3 Block computations

An interesting class of systems is the class of systems that perform block computations. This thesis focuses on these block computation systems and more specific on block sorters.

Block computations work on blocks, where a block consists of a number of successive values from a stream. For each stream, the number of values that form a block, referred to as the block size, is fixed. Blocks can be numbered which allows us to speak of the block with block number 0 or 5. The values in a block with block number n from an output stream only depend on the values in the block with block number n from the input streams. This means that there are a finite number of input values that have an influence on an output value.

Formally, let S be a stream of values whose k^{th} value is $S[k - 1]$ and let N be the block size of stream S , then block S_i from stream S with block number i is defined as $S[i * N .. (i + 1) * N)$. Moreover, the values in each block can be indexed. Let $S_i[j - 1]$ be the j^{th} value of the block with block number i from stream S , then we have $S_i[j] = S[i * N + j]$, for $0 \leq j < N$. Finally, let $MS(S_i)$ be the multi-set (sometimes called bag) of values in S_i .

This means that if a system with input stream A and output stream B performs a block computation, then the j -th value of the i -th block of the output stream B is equal to a function f_j applied to the values of the i -th block of the input stream A :

$$B_i[j] = f_j(A_i)$$

For instance, we look at two periodic stream operators, the take and drop operators [6] that can be used to perform block computations. The take operator \mathcal{T}_l^{j-1} that has periodicity l and rank $j - 1$ operating on a stream A results in a stream consisting of every j^{th} of l values of stream A . In terms of block sizes, stream A has block size l and the resulting stream has block size 1.

The drop operator \mathcal{D}_l^{j-1} that has periodicity l and rank $j - 1$ operating on a stream A results in a stream consisting of all but every j^{th} of l values of stream A . In terms of block sizes, stream A has block size l and the resulting stream has block size $l - 1$.

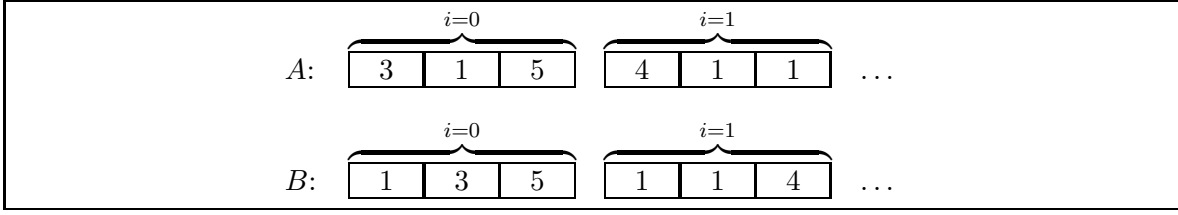


Figure 1.1: Example in- and output streams for a block sorter, sorting blocks of 3 values.

Formally \mathcal{T}_l^j and \mathcal{D}_l^j are defined by the concatenation ($++$) of values from stream A in such a way that:

$$\mathcal{T}_l^j A = (++ i : 0 \leq i \wedge i \bmod l = j : A[i]) \quad (1.1)$$

$$\mathcal{D}_l^j A = (++ i : 0 \leq i \wedge i \bmod l \neq j : A[i]) \quad (1.2)$$

An example system that performs a block computation is the system with input stream A and output streams $B = \mathcal{T}_2^0 A$ and $C = \mathcal{T}_2^1 A$.

Block sorters

Let $\overline{A[l..u]}$ denote the sequence that results from sorting $A[l..u]$. To be precise $\overline{A[l..u]}$ is the unique sequence such that

$$\left(\forall j, k : 0 \leq j < k < u-l : \overline{A[l..u]}[j] \leq \overline{A[l..u]}[k] \right) \wedge MS(\overline{A[l..u]}) = MS(A[l..u]) \quad (1.3)$$

A block sorter with block size N is a system with input stream A and output stream B that satisfies the functional specification

$$(\forall i : 0 \leq i : B_i = \overline{A_i})$$

Recall that the meaning of B_i depends on the block size N .

Hence, for a block sorter, the output blocks are the sorted input blocks. Figure 1.1 is an example which shows an input stream A and the corresponding sorted output stream B for a block sorter system sorting blocks of 3 values.

Moreover, we introduce an operator \bowtie that merges two sorted streams into one sorted stream, defined by

$$\overline{A[l..l+m]} \bowtie \overline{A[l+m..l+2m]} = \overline{A[l..l+2m]}$$

1.2 Problem description

The previous section described the systems that are considered in this thesis. This section discusses the research questions that have been investigated for these systems.

This thesis is **not** about designing stream processing systems. The construction of such systems from a formal specification is outside the scope of the thesis. Amongst others this means that given a system we take its functional correctness for granted or at its best provide a few informal arguments.

However, when a system is derived from its formal specification, design decisions are made. Not only do these decisions guarantee the functional correctness of the system, they also influence other aspects of the resulting system, such as cost, testability and performance. Some of these aspects can be quantified and captured by metrics. When such a metric describes part of the performance of such a system, we call it a performance metric.

When considering practical applications of stream processing systems, there are several possible designs per systems that meet the same functional requirements, but with different values for the metrics. For any particular application some metrics might be considered to be of more importance than others. Hence choosing the ‘right’ design for an application is a matter of comparing the possible designs with respect to the values of the important metrics.

Only when **SPSs** are realized, for instance in hardware, is it possible to *measure* the values for performance metrics. However, it is possible to create and use tools that predict values by using (mathematical) models. With these models, it is possible to compare designs without having to actually build the designs, greatly reducing the cost of the design cycle.

The main goal of the research for this thesis is analyzing performance metrics with a mathematical model for **SPSs** . Determining values for performance metrics in an analytical manner would be very nice, however this can not be done for all designs of all systems. This is why we look at another approach, constructing and solving optimization problems, to establish these values.

1.2.1 Performance analysis

One of the quantities that play an important role in the performance analysis of systems is time, be it in the form of the time it takes for events to happen (e.g. response time, cycle time, latency) or the number of events per unit of time (e.g. throughput). In this thesis we use a timing model that is discrete, meaning that time is divided into time slots. In this time model, when an event takes place in a time slot, it starts at the beginning of that time slot and is completed at the end of the time slot.

To ease the introduction of some of these metrics, we introduce a metaphor that describes communication in terms of transportation of tokens through systems.

Imagine that values are written on little balls which we will call tokens. For components reading a value from an input port and storing it in a variable, this means accepting the token and putting it in storage. For components writing a value to an output port, this means removing the token from storage and delivering it. There is a difference between the metaphor and the actual case however. In the metaphor, the token is delivered and gone from the component while the variable actually still contains the last read value.

A communication action between an input port and an output port is the transportation of the token from the component at the input side of the channel to the component at the output side of the channel. Systems accept tokens from the environment at the input side and they deliver tokens to the environment at the output side. Once a token is accepted from the environment, it is present in the system until that token is delivered to the environment. Moreover it is possible to count tokens that are in the system.

Note that the above only describes transportation of tokens through the system. Computations can be described by reading and altering the values written on tokens. In general, this metaphor may not be able to describe all possible computations, but it is sufficient for the scope of this thesis.

Before we state the goal of our research, we first introduce some performance metrics.

Capacity

The maximum number of tokens that can be present in a system at any given time is called the capacity of a system.

A one-place buffer for example, can store at most one token, it has capacity 1. This can be seen by the fact that if there is a token present in the buffer, it has to deliver that token to its environment before the buffer can accept a new token.

Average i/o-distance

The average number of storage locations that a token passes on its way through a system is called the average i/o-distance of that system. For a system consisting of a single component, the average i/o-distance is 1, for larger systems, it is the average number of components that tokens pass on their path through the system.

Average throughput

Average throughput is the average rate at which a system sends data to (or receives data from) its environment, or stated otherwise, the average number of tokens accepted from or produced by the system per time slot.

For example, consider a one-place buffer component that accepts a token from the environment every three time slots and delivers it to the environment one time slot after accepting it. Since it accepts one token every three time slots, its throughput equals $\frac{1}{3}$. Note that for systems that can store only a finite number of tokens, we must obtain the same average throughput whether we look at the input or output side of the system.

Average cycle time

The average cycle time is the inverse of the average throughput. This means it is a measure of the time that elapses between sending data-items to (or receiving data-items from) its environment.

For example, a buffer running at average throughput $\frac{1}{3}$ has average cycle time 3.

Average occupancy

The occupancy of a system tells something about how many tokens are present at that system at a given time slot and it is defined as the number of tokens present at the end of that time slot. Obviously, the occupancy at any given time slot is between 0 and the capacity of the system. The average occupancy for a number of time slots is calculated by taking the average of all occupancies for those time slots.

For instance, consider a one-place buffer component that is working at its maximum throughput. This means that the buffer is working constantly, accepting a token from its environment in one time slot and immediately delivering it to its environment in the next time slot after which this process repeats in the next time slot. If we count the number of tokens in the buffer component at the end of all time slots, we see that there is one token present half of the time and no token present the other half of the time, so the average occupancy of the one-place buffer component working at maximum throughput is $\frac{1}{2}$.

Average latency

The latency of a system is a measure of the delay between the time a system accepts a token from its environment and the time the token is delivered to the environment again. Since not all tokens take the same path through the system, we use average latency to measure the average time tokens are in the system.

Again we look at a one-place buffer component. If it accepts a token from the environment in a time slot, it can deliver that token to the environment one time slot after accepting it at the earliest, meaning that the minimal average latency is 1. However, it can also deliver the token to the environment at the latest moment, that is one time slot before accepting a new token from its environment. This gives the maximum average latency.

Elasticity

One particular interesting metric for stream processing systems is the elasticity of the system. The elasticity of a system is a metric that expresses how well the system can maintain its maximal throughput under varying average occupancies of the system. A system with a higher elasticity can handle temporary differences in the rate at which the environment delivers inputs or accepts outputs better.

Elasticity can be determined by looking at the difference between minimal and maximal latency or minimal and maximal occupancy.

1.2.2 Project goal

The metrics in the previous section, with the exception of the capacity and the i/o-distance, depend on schedules. This means that schedules play an important role in the performance analysis of systems.

A schedule is a mapping of all the communication actions of a system onto discrete time slots. For performance analysis, some schedules are more interesting than others. Schedules that achieve minimal or maximal average latency of the system for a given average throughput, for instance, are very interesting. These types of schedules that achieve either minimal or maximal average latency for a given average throughput are called the *extreme schedules* of a system.

In general, extreme schedules can not be found analytically. However, extreme discrete time schedules for a stream processing system can be calculated by formulating and solving an integer linear programming (ILP) [9] problem, as we show in Chapter 5.

The goal of this project is to facilitate the performance analysis of stream processing systems with synchronous communications and data-independent communication behavior by

- providing a construction that for a given **SPS** with data-independent communication behavior produces ILP problems whose solutions provide the above mentioned extreme schedules;
- providing tool support to automatically generate these ILP problems in a format that can be used in an of-the-shelf ILP solver to generate a solution;
- applying this tool to various block sorter designs to determine their elasticity.

1.3 Document overview

The remainder of this thesis starts with some theoretical background, divided into four chapters. Chapter 2 deals with the structure of the systems under consideration. It defines the components that are the basic building blocks of these systems and the manner in which they can be composed to form new systems. Chapter 3 deals with the behavioral aspects of these systems such as events and their scheduling. Chapter 4 details the performance of stream processing systems. Important topics handled in that chapter are definitions of performance metrics and dependencies between these metrics. Chapter 5 describes integer linear programming problems. It defines the manner in which these programming problems are constructed from the program text of a system. Chapter 6 describes the tool that we have developed to synthesize such integer linear programming problems from the systems' textual representation. Chapter 7 thesis results produced when applying this tool to various block sorter designs. Chapter 8 contains conclusions and suggestions for further research.

Chapter 2

System structure

This chapter discusses, in an informal but precise way, the structure of the systems that are investigated in this thesis. Most of the notions in this chapter are presented by giving examples that illustrate the most important aspects. The formal syntax of these systems, necessary for the development of the tool, is given in Appendix A.

Systems that we consider are constructed in a compositional manner, which means that we have systems that are ‘smallest’, in the sense that they are indivisible, and a way to combine systems to form larger systems.

We call these indivisible systems *components*. Components are the topic of Section 2.1. The method to define new systems by composing already defined systems is the topic of Section 2.2.

2.1 Basic building blocks

We have a large number of different components to our disposal. However, we only introduce the components we use in this thesis. We focus on systems that perform a very common task, sorting blocks of values and these systems are called block sorter systems. Block sorter systems *can* be constructed with four types of components, however, a fifth type of component, the first-merge-last-split component, is also introduced because of its superior performance.

The components we introduce in this section are the one-place buffer component, the merge component, the split component, the first-merge-last-split component and the comparator component. The first-merge-last-split component is a combination of one merge and one split component and it has better performance than these two components combined.

The components are defined in a hardware definition language, similar to Haste [8] and Balsa [1]. The complete grammar for the language is given in Appendix A. The component declarations and descriptions in this section are taken from [7].

The figures in this section contain the program texts for the components, as well as a diagram. Although the complete definition of a system is in the program text, the structure of (especially large) systems is usually better understood when using a diagram, so we present both the program text and a diagram.

The first component introduced is the one-place buffer. Its program text and the diagram are given in Figure 2.1. Its definition consists of a heading and a body. The heading specifies that we are defining a component (by using keyword **comp**) with name *Buf* that has one input port *a* along which it receives stream *A* of input values from its environment. Furthermore,

component *Buf* has one output port *b* along which it sends stream *B* of output values to its environment. Viewed as a stream transformer, a one-place buffer, like all other buffers, performs the identity transformation, that is output stream *B* equals input stream *A*.

The body consists of a declaration of a single variable *x* in which the component can store values, followed by a command. The command defines the order of communication events in which the component is involved. In the case of *Buf* it expresses that there is an infinite repetition (denoted by $\#[\dots]$) of an input action on port *a* that stores the value read in variable *x* (denoted by $a?x$), followed by (denoted by the sequential composition operator ‘;’) an output action on port *b* of the value of *x* (denoted by $b!x$).

```
Buf = comp ( in a & out b ).
| [ var x
|> #[ a?x ; b!x ]
| ]
```

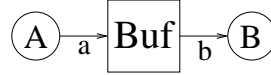


Figure 2.1: A one-place buffer.

The second component is the split component, $Split_l^k$, which is defined in Figure 2.2. In the heading of the declaration of *Split* there is one input port *a* along which it receives the stream *A* of input values, two output ports *b* and *c* along which it sends the streams *B* and *C* respectively of output values and furthermore it has two value parameters *k* and *l* denoted by keyword **val**. The value parameters provide a way to define a large class of similar components by using a sort of template. The actual values for these parameters are static and must be known at ‘compile’ time.

The diagram of the $Split_l^k$ in Figure 2.2 contains a marker at output port *b*. This marker marks the output port where output number *k* is generated. This convention allows us to leave out channel names in diagrams.

The functionality of the split component in terms of streams *A*, *B* and *C* is that of every block of *l* values from the input stream *A*, it sends all but the $(k+1)^{th}$ value along stream *C* and the $(k+1)^{th}$ value along stream *B*. In other words, $Split_l^k$ satisfies specification $B = \mathcal{T}_l^k A$ and $C = \mathcal{D}_l^k A$.

The command in the body of the declaration of $Split_l^k$ expresses that there is an infinite repetition of *k* times (denoted by $\#k[\dots]$) one input action on port *a* ($a?x$) and one output action on port *c* ($c!x$), followed by one input action on port *a* and one output action on port *b*, again followed by $l - k - 1$ times (denoted by $\#(l - k - 1)[\dots]$) one input action on port *a* and one output action on port *c*.

```
Split = comp ( in a & out b, c & val k, l ).
| [ var x
|> #[
|     #k[ a?x ; c!x ]
|     ; ( a?x ; b!x )
|     ; #(l-k-1)[ a?x ; c!x ]
| ]
| ]
```

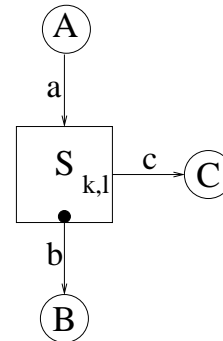


Figure 2.2: A split component.

Third, we have the merge component, $Merge_l^k$ (defined in Figure 2.3) which is similar to the $Split_l^k$ component. The functionality of $Merge_l^k$ in terms of streams A , B and C satisfies specification $A = \mathcal{T}_l^k C$ and $B = \mathcal{D}_l^k C$, meaning that for every l values it sends along stream C , it receives all but the $(k + 1)^{th}$ value from stream B and the $(k + 1)^{th}$ value from stream A .

```

Merge = comp ( in a, b & out c & val k, l ).
|[ var x
[> #[
    #k[ b?x ; c!x ]
    ; ( a?x ; c!x )
    ; #(1-k-1)[ b?x ; c!x ]
    ]
]|

```

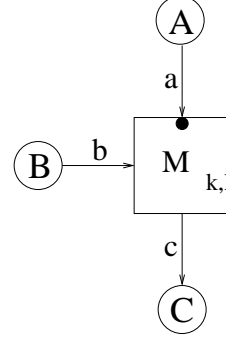


Figure 2.3: A merge component.

The fourth component is the comparator or compare-and-swap component, $Comp$, defined in Figure 2.4. With two input ports (a and b for receiving streams A and B respectively) and two output ports (c and d for sending streams C and D respectively), the specification of the functionality in terms of streams A , B , C and D is $C = A \downarrow B$ and $D = A \uparrow B$.

The body of the comparator components declares two variables x and y in which the comparator can store values. Furthermore the command expresses that in an infinite repetition two concurrent input actions (denoted by the concurrency-operator ‘,’ which expresses that the actions can be executed in any order or even concurrently) $a?x$ and $b?y$ are followed by two concurrent output actions that send the values of the expressions $\min(x, y)$ and $\max(x, y)$.

```

Comparator = comp ( in a, b & out c, d ).
|[ var x, y
[> #[ a?x, b?y ; c!min(x,y), d!max(x,y) ]
]|

```

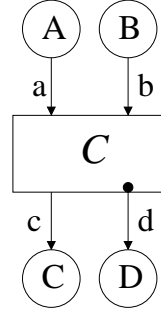


Figure 2.4: A comparator component.

The last component that we define, is the first-merge-last-split component $FMLS_l$ defined in Figure 2.5. It has two input ports a and b along which it receives streams A and B respectively and two output ports c and d along which it sends streams C and D respectively. As the name suggests, the functionality of the $FMLS_l$ component is the combination of the functionality of a $Merge_l^0$ and $Split_l^{l-1}$ component. Hence from a functional point of view, it is superfluous. However, by combining the two separate components in one larger component, one variable can be eliminated so it requires less storage space. Moreover $FMLS_l$ components are faster, since fewer variables induce less latency. The functional specification for the inputs of the system is that for every l values received by the component, the first value is received

on port a and the remaining $l-1$ values are received on port b . For the outputs of the system, we see that of every l values sent by the component, the first $l-1$ values are sent over port d and the last value is sent over port c . In terms of input streams A and B and output streams C and D , this is formalized by the following equations $C = \mathcal{T}_{l-1}^{l-2}B$, and $\mathcal{T}_{l-1}^0D = A$, and $\mathcal{D}_{l-1}^0D = \mathcal{D}_{l-1}^{l-2}B$.

```

FMLS = comp ( in a, b & out c, d & val l ).
|[ var x
|> #[
      a?x
      ; #(l-1)[ d!x ; b?x ]
      ; c!x
    ]
]|

```

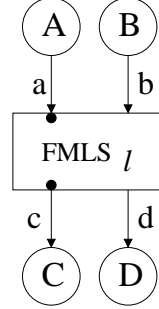


Figure 2.5: A first-merge-last-split component.

Note that program texts specify *all* aspects of stream processing systems: structure, functionality and behavior, i.e., the order in which events of the system can take place.

2.1.1 Component attributes

Given a definition for a component, or later on a system, by means of a program text, various derived entities can be defined. For example, think of the set of names of all variables, the list of input ports for a component, etcetera. In the remainder of this thesis, these entities are called attributes, because they can be defined by using an attribute grammar to describe the formal syntax of components.

Attributes serve two important goals. In the first place, they impose extra conditions on the context that have to be met for a program text to define a valid component. For example, a component with one input port named a can not have another in- or output port named a . Secondly, some of these attributes are used in the definition of performance metrics. All attributes can be synthesized from declarations of components. The ones that impose further restrictions on the definition of the component are mentioned in Appendix A, some of the others, that are used to describe the behavior of a component are given below.

A very obvious, however important, attribute is the name of the component. Another attribute that we can generate is the port set (\mathcal{PS}) of a component.

The port set of component X consists of the input port set of X ($\mathcal{PS}_{in}(X)$) and the output port set of X ($\mathcal{PS}_{out}(X)$), thus $\mathcal{PS}(X) = \mathcal{PS}_{in}(X) \cup \mathcal{PS}_{out}(X)$. The input port set of X contains the names of the input ports of component X , the output port set of X contains the names of the output ports of X . For example $\mathcal{PS}(Buf) = \mathcal{PS}_{in}(Buf) \cup \mathcal{PS}_{out}(Buf) = \{a\} \cup \{b\} = \{a, b\}$.

Similar, we have the value set of component X ($\mathcal{ValS}(X)$) that contains the names of the value parameters of component X , and the variable set of component X ($\mathcal{VarS}(X)$) that contains the names of the variables declared in component X .

We have implicitly made some assumptions with respect to the validity of declarations of components. With the attributes defined above, we make some of these assumptions explicit.

- Names for input and output ports must differ, thus for component X :
 $\mathcal{PS}_{in}(X) \cap \mathcal{PS}_{out}(X) = \emptyset$.
- Names for variables and value parameter must differ, thus for component X :
 $\mathcal{ValS}(X) \cap \mathcal{VarS}(X) = \emptyset$.
- Components have at least one port, thus for component X :
 $\mathcal{PS}(X) \neq \emptyset$.

A complete list of attributes is given in Appendix A.

2.2 System composition

As stated in the previous section, the first-merge-last-split component is a ‘combination’ of a merge and a split component. Therefore we start with an example that composes a system with the same functionality as component $FMLS$ from a $Merge_l^0$ and a $Split_l^{l-1}$ component. It is defined in Figure 2.6. The declaration of a system looks similar to the declaration of a component. The heading is the same only instead of the keyword **comp** we use the keyword **syst**. The body of the system declaration consists of a declaration of an internal channel e , followed by an instantiation of a *Merge* component, combined (denoted by the composition operator ‘||’) with an instantiation of a *Split* component. The instantiation of the *Merge* component consists of the name of the component (*Merge*) followed by a set of actual port names (a , b and e) and values for the formal parameters of the component (0 and l).

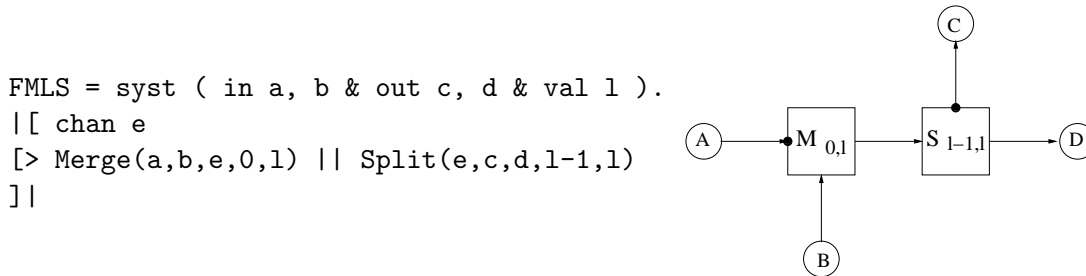


Figure 2.6: A first-merge-last-split system.

Another example of a composite system is the block sorter of block size 2 defined in Figure 2.7. This example uses four internal channels c , d , e and f , and combines three instances of components, viz. an instance of a *Split* component, an instance of a *Merge* component and an instance of a *Comparator* component.

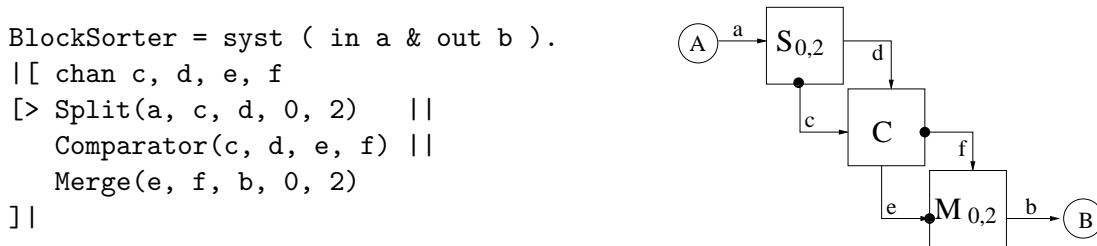


Figure 2.7: A block sorter system with block size 2.

Besides the construction of arbitrary parallel systems, we are often confronted with the construction of a set of systems that belong to the same class, i.e., a set of systems that have a similar structure or perform a similar task. An example of such a class of systems is the class of block sorters where we have a separate system for each block size N .

Another example of such a class is the class of linear buffers. The linear buffers with capacity and 2 and 3 are shown in Figures 2.8 and 2.9.

```
LBuf2 = syst ( in a & out b ).
|[ chan c
[> Buf(a, c) || Buf(c, b) ]|
```

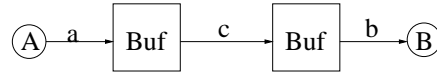


Figure 2.8: Linear buffer systems: LBuf2.

```
LBuf3 = syst ( in a & out b ).
|[ chan c
[> LBuf2(a, c) || Buf(c, b) ]|
```

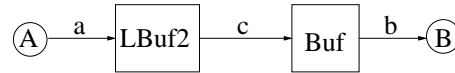


Figure 2.9: Linear buffer systems: LBuf3.

We are working towards a recursive definition of this class of linear buffers. We see that LBuf2 consists of two normal buffer components, while the LBuf3 consists of a normal buffer component and a smaller linear buffer system. It is nicer if we could define LBuf2 in such a way that it is composed of a normal buffer component and a smaller linear buffer system. This makes the definition of the class more uniform.

We note that we view a single component as a system and while this is often implicit, it can be made explicit. An example is LBuf1 from Figure 2.10. We can use this definition of LBuf1 in the definition of LBuf2 so that it also consists of a normal buffer component and a smaller linear buffer system.

We introduce a template-like language construct, the *case statement* to further facilitate the recursive definition of systems. With the case statement, the actual definition of a system at compile time is dependant on an expression that can be based on parameter values. Hence, with the case statement it possible to recursively define systems. We further demonstrate this with an example in Figure 2.11.

```
LBuf1 = syst ( in a & out b ).
|[ Buf(a, b) ]|
```

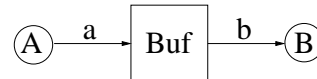


Figure 2.10: Linear buffer systems: LBuf1.

```
LBuf = syst ( in a & out b & val n ).
case n = 1 -> |[ Buf(a, b) ]|
[] n > 1 -> |[ chan c
                [> Buf(a, c) ||
                 LBuf(c, b, n-1)
                ]|
esac
```

Figure 2.11: Linear buffer systems, recursive declaration.

With this case-construct, it is possible to recursively define classes of systems, for instance the class of linear buffers from Figure 2.11. However, we keep it very simple, meaning that we require that the values for these parameters must be known at compile time, so that we can unfold the definition. This means that after the program text is parsed, we can flatten all these case constructions.

2.2.1 System attributes

As with components, it is possible to synthesize attributes from the declaration of a system. We have the port set of a system Sys , denoted $\mathcal{PS}(Sys)$. Note that once a system is instantiated, formal parameters are replaced by actual names and the port set reflects this. So, for instance, $\mathcal{PS}(Buf(c, d)) = \mathcal{PS}(Buf)_{[a \leftarrow c, b \leftarrow d]} = \{c, d\}$.

Besides the port set of a system Sys , we also have the channel set (denoted by $\mathcal{CS}(Sys)$) containing the names of all the channels of Sys , both channels declared in Sys and channels declared in any subsystem of Sys . Since a system does not always have internal channels, for instance when the system is a component, the channel set can be empty.

We introduce the notion of the alphabet of system Sys , denoted $\mathcal{A}(Sys)$ that contains the names of the channels and ports of system Sys , so $\mathcal{A}(Sys) = \mathcal{PS}(Sys) \cup \mathcal{CS}(Sys)$.

There is a potential danger of name clashes when composing systems. Take for instance $LBuf3$ from Figure 2.9 which has an internal channel named c and consists of the composition of instantiations of a Buf component and a $LBuf2$ system. System $LBuf2$, however, also has an internal channel named c , which means that name c is no longer unique. This means that if we calculate \mathcal{CS} by simply combining the channel sets of the composed systems, then this yields the wrong answer, since we cannot distinguish between channel c from system $LBuf3$ and channel c from system $LBuf2$.

$$\mathcal{CS}(LBuf3) = \{c\} \cup \mathcal{CS}(LBuf2) \cup \mathcal{CS}(Buf) = \{c\} \cup \{c\} \cup \emptyset$$

In the sequel, we assume an implicit renaming of internal channels. Here however, we make this renaming explicit once to show how name clashes can be avoided. We do so by giving a name to the instantiations of systems before we allow them to be composed (see Figure 2.12).

Remark that in this example, we have flattened the case constructions to calculate the channel set.

If we prefix the channel names of subsystems with the name of these subsystems, we get unique names for all channels. Using this scheme for $LBuf3$ we have find that

$$\begin{aligned} \mathcal{CS}(LBuf3) &= \{c\} \cup LB2.\mathcal{CS}(LBuf2(a, c)) \cup B.\mathcal{CS}(Buf(c, b)) \\ &= \{c, LB2.c\} \cup LB2.LB1.\mathcal{CS}(LBuf1(a, LB2.c)) \cup LB2.B.\mathcal{CS}(Buf(LB2.c, b)) \\ &= \{c, LB2.c\} \end{aligned}$$

Another possibility to avoid these name clashes is to use a global counter. Whenever we encounter a declaration of a channel, we substitute the name of that channel with the next unique value of the counter in the entire scope that the channel name is valid in. This technique can easily be performed in the same step as the flattening of the definition of the system after parsing the program text.

```
LBuf1 = syst ( in a & out b ).  
|[ B = Buf(a, b)  
[> B ]|
```

```
LBuf2 = syst ( in a & out b ).  
|[ chan c  
; LB1 = LBuf1(a, c)  
; B = Buf(c, b)  
[> LB1 || B ]|
```

```
LBuf3 = syst ( in a & out b ).  
|[ chan c  
; LB2 = LBuf2(a, c)  
; B = Buf(c, b)  
[> LB2 || B ]|
```

Figure 2.12: Linear buffer systems, defined such that name clashes are avoided.

Chapter 3

System behavior

This chapter discusses the behavioral aspects of systems. To that end it introduces two new system attributes, viz. events and schedules. Like the attributes from the previous sections, that deal with structural aspects of **SPSs**, they can be defined by induction over the structure of a system.

3.1 Events

When a system is executed, every communication statement (also called an action) of a component of that system gives rise to zero or more (possibly infinite) events. A distinction is made between internal events, that involve the cooperation of two system components, and external events, that involve the cooperation of one component of the system with the environment.

An event is identified by a name and a non-negative occurrence number. In case of an external event, the name is a port name, and in case of an internal event, the name is a channel name. So, formally, an event e of system X is an element of $\mathcal{A}(X) \times \mathbb{N}$. For instance, the events associated with the instance $LBuf2(d, e)$ of Figure 2.8 are:

$$\{c\#i|i \leq 0\} \cup \{d\#i|i \leq 0\} \cup \{e\#i|i \leq 0\}$$

Events can be made explicit in the program text by the introduction of ghost variables that indicate occurrence numbers. As an aside, we remark that with the occurrence number present in the program text, it is often easier to show the functional correctness by using appropriate annotation of the program text. Figure 3.1 shows a version of the *Buf* component annotated using a ghost variable i . In the program text we have replaced all the port names by the events that occur at that port. Moreover we have annotated the program with assertions that relate the values of the program variables to the data values of the process streams. For instance, after communication action $a\#i?x$ we know that variable x has value $a(i)$. Moreover, the assertion after event $b\#i$ in the annotation shows that $b(i) = a(i)$, which is the required behavior of a buffer. However, as previously stated, we take the functional correctness of the systems for granted.

For system X we define the set $\mathcal{E}(X)$ of the events of X by induction over the structure of X . We give the set of events of the five basic components as base cases and use structural induction to define the set of events for composite systems.

```

Buf = comp ( in a & out b ).
|[ var x ; ghostvar i
[> i := 0 ;
  #[ a#i ? x ; { x = a(i) }
    b#i ! x ; { b(i) = a(i) }
    i := i + 1
  ]
]|

```

Figure 3.1: A one-place-buffer, defined with occurrence number i .

- $\mathcal{E}(Buf) = \{a\#i|i \leq 0\} \cup \{b\#i|i \leq 0\}$
- $\mathcal{E}(Split) = \{a\#i|i \leq 0\} \cup \{b\#i|i \leq 0\} \cup \{c\#i|i \leq 0\}$
- $\mathcal{E}(Merge) = \{a\#i|i \leq 0\} \cup \{b\#i|i \leq 0\} \cup \{c\#i|i \leq 0\}$
- $\mathcal{E}(Comp) = \{a\#i|i \leq 0\} \cup \{b\#i|i \leq 0\} \cup \{c\#i|i \leq 0\} \cup \{d\#i|0 \leq i\}$
- $\mathcal{E}(FMLS) = \{a\#i|i \leq 0\} \cup \{b\#i|i \leq 0\} \cup \{c\#i|i \leq 0\} \cup \{d\#i|0 \leq i\}$

For system Z with port set $\mathcal{PS}(Z)$ and channel set $\mathcal{CS}(Z)$, which consists of instantiations of systems X and Y (abbreviated $Z = X||Y$), we have that $\mathcal{E}(Z) = \mathcal{E}(X) \cup \mathcal{E}(Y)$.

3.2 Precedence graphs

Apart from the events as such, a program text also (partially) specifies an order in which these events have to be executed. How this works formally, is explained in [13].

Informally, this means that we have a relation between two events e and f , $e \rightarrow f$, which states that event e *directly precedes* event f , which follows from the programming text because the statements responsible for those events are separated by precisely one semicolon. Note that we have ‘implicit’ semicolons, for instance between two iterations of a repetition, for instance in the *Buf* component we have $\sigma(b\#i) \rightarrow \sigma(a\#(i+1))$. Furthermore, we note that relation \rightarrow is not a transitive relation. This can be remedied by taking the transitive closure of \rightarrow , which gives us a relation \rightarrow^+ , with the following properties: $a \rightarrow b \implies a \rightarrow^+ b$ and $a \rightarrow^+ b \wedge b \rightarrow^+ c \implies a \rightarrow^+ c$.

Using the directly precedes relation, it is possible to construct a precedence graph for a component. A precedence graph of a component is a directed graph with the events of the component as its nodes; the edges of the graph are given by the directly precedes relation, \rightarrow , for the component. This means that there is a directed edge from the node of event e to the node of event f , if and only if $e \rightarrow f$. Stated formally, the precedence graph of component C is $G(C) = (\mathcal{E}(C), \rightarrow)$. Now, if there is a directed path from event e to event f in the precedence graph, we say that e precedes f , denoted $e \rightarrow^+ f$.

For example, if we look at a one-place buffer component, we see that the directly precedes relation is the smallest relation that satisfies $(\forall i : 0 \leq i : a\#i \rightarrow b\#i)$ and $(\forall i : 0 \leq i : b\#i \rightarrow a\#(i+1))$. Hence, in the precedence graph of a one-place buffer component, we have nodes

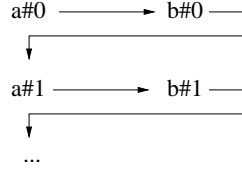


Figure 3.2: Precedence graph for a one-place buffer component.

$a\#i$ and $b\#i$, for $0 \leq i$. Moreover, we have edges from $a\#i$ to $b\#i$ and edges from $b\#i$ to $a\#(i+1)$ for $0 \leq i$ (see Figure 3.2).

Note that there is no edge from $a\#0$ to $a\#1$ in the precedence graph of the one-place buffer, since in this case $a\#0$ does not directly precede $a\#1$. However, because there is a directed path from $a\#0$ to $a\#1$ in the precedence graph, we can still conclude that $a\#0$ precedes $a\#1$.

Precedence graphs are usually fairly simple, for instance precedence graphs for components that only contain sequential communication actions are just linear chains. Because relation \rightarrow is irreflexive, the precedence graph of a single component does not contain any auto-cycles. However, it is possible that when components are combined to form systems, cycles are introduced in the precedence graph. These cycles are unwanted, because if a precedence graph contains a cycle, the system contains a deadlock, since having a cycle means that there is an event that precedes itself and we intuitively know that this cannot be the case. We prove this in Section 3.4.

The directly precedes relation for a comparator component is the smallest relation that satisfies both

$$(\forall i : 0 \leq i : a\#i \rightarrow c\#i \wedge a\#i \rightarrow d\#i \wedge b\#i \rightarrow c\#i \wedge b\#i \rightarrow d\#i)$$

and

$$(\forall i : 0 \leq i : c\#i \rightarrow a\#(i+1) \wedge c\#i \rightarrow b\#(i+1) \wedge d\#i \rightarrow a\#(i+1) \wedge d\#i \rightarrow b\#(i+1))$$

The precedence graph of the comparator looks somewhat different from the precedence graph for the one-place buffer component. Because of the concurrent behavior of this component, we see that every node, except the first two, has two incoming and two outgoing edges, shown in Figure 3.3.

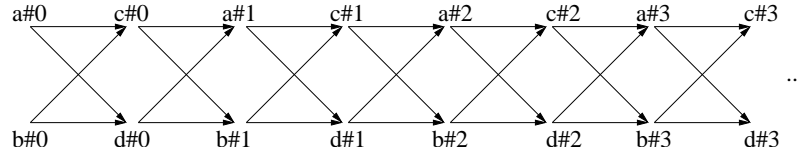


Figure 3.3: Precedence graph for the comparator component.

Although precedence graphs consist of an infinite number of nodes, we usually consider only a finite subgraph of these precedence graphs. Such a finite subgraph, which we call a downward closed subgraph, captures the initial behavior of a system. This means that for the downward closed subgraph $G1$ of graph G , if an event $a\#i$ on port a is a node of $G1$, then all events on port a preceding $a\#i$ are nodes of $G1$ and all the edges $v_1 \times v_2 \in G$ where

$v_1 \in G_1$ and $v_2 \in G_1$ are also edges of G_1 . Formally, subgraph $G_1 = (V_1, E_1)$ is a downward closed subgraph of graph $G = (V, E)$ if

$$i < j \wedge a\#j \in V_1 \implies a\#i \in V_1$$

and

$$E_1 = E \cup (V_1 \times V_1)$$

We can construct downward closed subgraphs by specifying for each port or channel of a system, the maximal occurrence number for events on that port or channel.

Furthermore, recall that we are constructing linear programming problems. These programming problems are finite, meaning that they require that the problem is expressed in a finite set of decision variables and constraints. This means that if we want to use this technique, we have to try to describe the infinite behavior of such systems in a finite manner and that is why we use downward closed subgraphs. We remark that we are dealing with components consisting of an infinite repetition of a finite sequence of communication actions. Because the sequence of communication actions is finite in these systems, they eventually exhibit periodic behavior. This means that any system will eventually return to a state that is similar to a state the system was in previously. This periodic behavior occurs in these systems consisting of a finite number of components, because these systems are finite state machines. With an infinite stream of input values, a system will eventually either enter a state of deadlock or return to a state it was in before. Since we do not consider systems that have deadlock (we assumed functionally correct systems), we have that the systems under consideration are periodic.

3.3 Derived precedence graphs

We are going to reduce the number of edges in a downward closed subgraph of the precedence graph. The goal is to reduce the number of constraints in the linear programming programs we are constructing, which hopefully lead to shorter computation times. To that end, we reason about the comparator component, and concurrent statements in components in general, some more. We try to figure out if there is a more intuitive way to look at such components and we construct a derived precedence graph.

If we consider the body of the program text of the comparator component, then we see that two concurrent communications are performed ($a?x$ and $b?y$), followed by two other concurrent communications ($c!\min(x,y)$ and $d!\max(x,y)$). The concurrency operator does not induce an ordering on events $\sigma(a\#i)$ and $\sigma(b\#i)$, however, we do know that both events have to be completed before events $\sigma(c\#i)$ and $\sigma(d\#i)$ take place. This is known as the distributed termination convention: an action that consists of several concurrent communication actions is terminated only when all concurrent communication actions are terminated.

Because of this distributed termination convention, there is a moment in time when both concurrent events $\sigma(a\#i)$ and $\sigma(b\#i)$ have terminated and neither events $\sigma(c\#i)$ and $\sigma(d\#i)$ has started. With this moment in time, we associate a distributed termination node. We view this node as a point in time at which the events are synchronized.

This means we introduce an extra type of node and an extra type of edge for our precedence graphs. The newly introduced type of node represents the distributed termination of a number of events, so it does not represent an actual event. The newly introduced type of directed

edge from node e to node f represents that e happens at the same time or later than f , so it does not represent an actual precedence relation. For each of the events associated with the concurrent actions, we have one edge of this type that ends in the distributed termination node. The derived precedence graph of a comparator component is depicted in Figure 3.4.

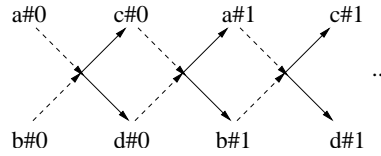


Figure 3.4: An derived version of the precedence graph of the comparator component.

In general, we introduce for $n > 1$ concurrent events, sequentially composed with $m \geq 1$ (possibly concurrent) other events, one distributed termination node and a number of n edges from the first set of events to that termination node. Furthermore, we introduce a normal edge from the termination node to the events in the second set and finally we remove all edges from the first set of events to the second set of events. By introducing one extra node, we have substituted $m * n$ edges by $m + n$ edges. We have that $2 \leq n$ (otherwise we would not have introduced a distributed termination node), furthermore we know that $2 \leq n \wedge 2 \leq m \implies m + n \leq m * n$, which means that unless $m = 1$, we have less edges in the derived precedence graph than we had before in the ‘normal’ precedence graph.

3.4 Event scheduling

In this section, we introduce the notion of a schedule. A schedule is mapping of the events of a component or system to discrete time slots. Not all mappings of events to discrete time slots form a schedule. Only mappings that respect the order of the precedence graphs qualify as a schedule. This means that if there is a directed path from event e to event f in the precedence graph, then the time slot at which event e takes place, is smaller than the time slot at which event f takes place. There can be a lot of those schedules for any component or system. Together, these schedules capture the behavior of a system. Some of them have certain nice properties, for instance minimal or maximal latency given a fixed average throughput. We are interested in schedules with those properties, because we can use them for performance analysis. Note that it is possible that there is no schedule for a system, which means the system contains a deadlock.

Various components of a system can be *active* simultaneously in one time slot, i.e., perform one or more events in that time slot. When a component is not active in a time slot, it is called *idle*.

Each communication action takes exactly one time slot to complete, meaning that events that originate from a single component can only be active in the same time slot if they are part of a concurrent statement.

As an example we take the comparator from Figure 2.4. While events $a\#2$ and $b\#2$ can take place in the same time slot (they are concurrent), events $a\#2$ and $c\#2$ cannot take place in the same time slot (they are sequential).

Formally, a schedule is a function σ with signature $\mathcal{E}(X) \rightarrow \mathbb{N}$. So, $\sigma(a\#3) = 17$ means that the event $a\#3$ occurs in time slot 17.

As for the ordering of events, we have for events e and f , if $e \rightarrow f$, meaning that the actions associated with these events are separated by exactly one semicolon, then by definition $\sigma(e) < \sigma(f)$. We use induction to show that $e \rightarrow^+ f \implies \sigma(e) < \sigma(f)$.

We have as a base case for $e \rightarrow^+ f$, that $e \rightarrow f$, for which we showed that $\sigma(e) < \sigma(f)$.

We have $e \rightarrow^+ f \implies \sigma(e) < \sigma(f)$ as induction hypothesis. We examine the case $g \rightarrow^+ f$ where for some event e , $g \rightarrow e \wedge e \rightarrow^+ f$:

$$\begin{aligned}
& g \rightarrow e \wedge e \rightarrow^+ f \\
\implies & \{\text{definition } \rightarrow\} \\
& \sigma(g) < \sigma(e) \wedge e \rightarrow^+ f \\
\implies & \{\text{Induction Hypothesis}\} \\
& \sigma(g) < \sigma(e) \wedge \sigma(e) < \sigma(f) \\
\equiv & \{\text{transitivity } <\} \\
& \sigma(g) < \sigma(f)
\end{aligned}$$

This means that:

$$(\forall e, f : e, f \in \mathcal{E}(S) : e \rightarrow^+ f \implies \sigma(e) < \sigma(f)) \quad (3.1)$$

We use this property to show that if a precedence graph for system S contains a cycle, meaning $(\exists e : e \in \mathcal{E}(S) : e \rightarrow^+ e)$, then there is no schedule for the system, so S has deadlock.

$$\begin{aligned}
& (\exists e : e \in \mathcal{E}(S) : e \rightarrow^+ e) \\
\implies & \{\text{Equation 3.1}\} \\
& (\exists e : e \in \mathcal{E}(S) : \sigma(e) < \sigma(e)) \\
\equiv & \{\text{irreflexivity } <\} \\
& \text{False}
\end{aligned} \quad (3.2)$$

The induced ordering on the events for a one-place buffer component, given by the precedence graph of the one-place buffer component (see Figure 3.2), means that a schedule for the one-place buffer component respects the inequalities from Equation 3.3.

$$\begin{aligned}
\sigma(a\#i) < \sigma(b\#i) & , 0 \leq i \\
\sigma(b\#i) < \sigma(a\#(i+1)) & , 0 \leq i
\end{aligned} \quad (3.3)$$

For every pair of events in relation \rightarrow , we have one inequality and every pair of events in relation \rightarrow corresponds to one edge in the precedence graph. This means that there is a one-to-one relation between these inequalities and the precedence graph of a component.

In Chapter 5 we are going to use integer linear programming (ILP) to find schedules with specific properties. ILP problems deal with linear equalities, whereas the formulas in Equation 3.3 provides us with linear inequalities. We introduce the notion of a slack variable to facilitate the formulation of the derived ILP problems dealing with equalities. Therefore, instead of looking at the time slots in which events take place, we look at the difference between the time slots in which two events take place (or the slack). We capture these differences in slack variables and if we require that the values for these slack variables are at least 1, we rewrite all the inequalities to equalities. This is possible because it holds that $\sigma(a\#i) < \sigma(b\#i) \equiv (\exists x_{a,b,i} : 0 < x_{a,b,i} : \sigma(b\#i) = \sigma(a\#i) + x_{a,b,i})$.

Note that we have introduced precisely one slack variable for each inequality, which means that every edge in the precedence graph is associated with exactly one positive slack variable. Furthermore, we remark that in this thesis, we look at infinite systems, but we use periodicity to keep the ILP problems finite.

Rewriting the inequalities from Equation 3.3 using slack variables that represent the time that passes between two events, yields the equalities from Equation 3.4. Note that we implicitly assume $0 \leq i$ for the occurrence number i of events and $1 \leq x$ for slack variables x , and unless these constraints differ, we do not include them in the formulas in the remainder of this thesis.

$$\begin{aligned}\sigma(b\#i) &= \sigma(a\#i) + x_{a,b,i} \\ \sigma(a\#(i+1)) &= \sigma(b\#i) + x_{b,a,i}\end{aligned}\tag{3.4}$$

Note that the names of the slack variables are chosen somewhat arbitrary. Further on in the thesis we abstract from these names, so we do not go into much detail about how they are chosen.

We point to the fact that since $\sigma(a\#(i+1))$ is expressed in terms of $\sigma(b\#i)$ and $\sigma(b\#i)$ in turn is expressed in terms of $\sigma(a\#i)$ we can express $\sigma(a\#(i+1))$ in terms of $\sigma(a\#i)$. We repeat these substitutions for all $i > 0$, which means that for the one-place buffer σ is defined by Equation 3.5.

$$\begin{aligned}\sigma(a\#i) &= \sigma(a\#0) + (\Sigma : 0 \leq j < i : x_{a,b,j} + x_{b,a,j}) \\ \sigma(b\#i) &= \sigma(a\#0) + (\Sigma : 0 \leq j < i : x_{a,b,j} + x_{b,a,j}) + x_{a,b,i}\end{aligned}\tag{3.5}$$

We have accomplished that we have transformed the precedence graph of a buffer component to a set of equalities that express all events of the buffer component in terms of the same initial event and a number of slack variables. We use these equalities in Chapter 5 in the construction of linear programming problems.

As another example we look at the $Split_l^k$ component. (Note that at ‘compile time’, parameters k and l have known values.) The split component has a periodicity of l , meaning that after l input actions and output actions, the component is back in its initial state. The block size of the input stream is l , the block size of output stream along port b is 1 and the block size of output stream along port c is $l-1$.

We have for $0 \leq q$:

$$\begin{aligned}\sigma(a\#q \cdot l + r) &< \begin{cases} \sigma(c\#(q \cdot (l-1) + r)) & \text{if } 0 \leq r < k \\ \sigma(b\#q) & \text{if } r = k \\ \sigma(c\#(q \cdot (l-1) + (r-1))) & \text{if } k < r < l \end{cases} \\ \sigma(b\#q) &< \sigma(a\#(q \cdot l + r + 1)) \\ \sigma(c\#q \cdot (l-1) + r) &< \begin{cases} \sigma(a\#(q \cdot l + r + 1)) & \text{if } 0 \leq r < k \\ \sigma(a\#(q \cdot l + r + 2)) & \text{if } k \leq r < l-1 \end{cases}\end{aligned}\tag{3.6}$$

If we know the values of the parameters k and l , we can rewrite the inequalities using slack variables the same way we did with the one-place buffer component. We give a more formal method to do this in Section 3.4.1.

Thus far, we have looked at the ordering of events in the case of sequential composition of communication actions. In the case of concurrent composition of communication actions, there is less ordering induced. For example, if we look at the comparator component, we can derive the inequalities from Equation 3.7 from the precedence graph of Figure 3.3.

$$\begin{array}{ll}
\sigma(a\#i) < \sigma(c\#i) & \sigma(c\#i) < \sigma(a\#(i+1)) \\
\sigma(a\#i) < \sigma(d\#i) & \sigma(c\#i) < \sigma(b\#(i+1)) \\
\sigma(b\#i) < \sigma(d\#i) & \sigma(d\#i) < \sigma(a\#(i+1)) \\
\sigma(b\#i) < \sigma(c\#i) & \sigma(d\#i) < \sigma(b\#(i+1))
\end{array} \tag{3.7}$$

By introducing slack variables for the comparator component like we did before with the one-place buffer, we rewrite these inequalities to the equalities of Equation 3.8 or equivalently, label all the edges from the precedence graph with slack variables.

$$\begin{array}{ll}
\sigma(c\#i) = \sigma(a\#i) + x_{a,c,i} & \sigma(a\#(i+1)) = \sigma(c\#i) + x_{c,a,i} \\
\sigma(d\#i) = \sigma(a\#i) + x_{a,d,i} & \sigma(b\#(i+1)) = \sigma(c\#i) + x_{c,b,i} \\
\sigma(c\#i) = \sigma(b\#i) + x_{b,c,i} & \sigma(a\#(i+1)) = \sigma(d\#i) + x_{d,a,i} \\
\sigma(d\#i) = \sigma(b\#i) + x_{b,d,i} & \sigma(b\#(i+1)) = \sigma(d\#i) + x_{d,b,i}
\end{array} \tag{3.8}$$

For the linear buffer, we saw that the schedule for an event can be expressed by a constant value ($\sigma(a\#0)$) plus a number of slack variables, all greater than 0. This does not trivially hold for the schedule of the comparator, however we can write input event $b\#0$ as $c\#0 - x_{b,c,0}$ which in turn we can write as $a\#0 + x_{a,c,0}$. This way, we can define all events of the comparator component in terms of a constant $a\#0$ and some slack variables. This is a rather straightforward way to generate the equalities for the comparator component. However, in Section 3.2 we also introduced derived precedence graphs by looking at distributed termination. With this distributed termination, we derive for the comparator component

$$\begin{array}{l}
(\exists w_{a,i}, w_{b,i} : 0 \leq w_{a,i} \wedge 0 \leq w_{b,i} \quad : \quad \sigma(a\#i) + w_{a,i} = \sigma(b\#i) + w_{b,i} \\
\quad \wedge \quad \sigma(a\#i) + w_{a,i} < \sigma(c\#i) \\
\quad \wedge \quad \sigma(a\#i) + w_{a,i} < \sigma(d\#i))
\end{array}$$

A similar reasoning holds for concurrent events $\sigma(c\#i)$ and $\sigma(d\#i)$. We introduce these slack variables w , which have bounds $0 \leq w$ (the bounds are not named explicitly in the remainder of this thesis).

More general, because of distributed termination convention, we can introduce for $0 \leq i < n$ concurrently executed read or write statements on ports p_i , which are therefore associated with events $\sigma(p_i)$ a total of n slack variables $0 \leq w_i$ and say that for all events $\sigma(p_i)$ and $\sigma(p_j)$ $\sigma(p_i) + w_i = \sigma(p_j) + w_j$.

We rewrite the inequalities from Equation 3.7 to those of Equation 3.9 with the extra constraints that $(\forall i : 0 \leq i : \sigma(a\#i) + w_{a,i} = \sigma(b\#i) + w_{b,i})$ and $(\forall i : 0 \leq i : \sigma(c\#i) + w_{c,i} = \sigma(d\#i) + w_{d,i})$.

$$\begin{aligned}
\sigma(a\#i) + w_{a,i} &< \sigma(c\#i) & \sigma(c\#i) + w_{c,i} &< \sigma(a\#(i+1)) \\
\sigma(a\#i) + w_{a,i} &< \sigma(d\#i) & \sigma(c\#i) + w_{c,i} &< \sigma(b\#(i+1)) \\
\sigma(b\#i) + w_{b,i} &< \sigma(c\#i) & \sigma(d\#i) + w_{d,i} &< \sigma(a\#(i+1)) \\
\sigma(b\#i) + w_{b,i} &< \sigma(d\#i) & \sigma(d\#i) + w_{d,i} &< \sigma(b\#(i+1))
\end{aligned} \tag{3.9}$$

By introducing slack variables x we rewrite these inequalities to equalities like we did before, leading to Equation 3.10.

$$\begin{aligned}
\sigma(c\#i) &= \sigma(a\#i) + w_{a,i} + x_{a,c,i} & \sigma(a\#(i+1)) &= \sigma(c\#i) + w_{c,i} + x_{c,a,i} \\
\sigma(d\#i) &= \sigma(a\#i) + w_{a,i} + x_{a,d,i} & \sigma(b\#(i+1)) &= \sigma(c\#i) + w_{d,i} + x_{c,b,i} \\
\sigma(c\#i) &= \sigma(b\#i) + w_{b,i} + x_{b,c,i} & \sigma(a\#(i+1)) &= \sigma(d\#i) + w_{c,i} + x_{d,a,i} \\
\sigma(d\#i) &= \sigma(b\#i) + w_{b,i} + x_{b,d,i} & \sigma(b\#(i+1)) &= \sigma(d\#i) + w_{d,i} + x_{d,b,i}
\end{aligned} \tag{3.10}$$

Since we required that $\sigma(a\#i) + w_{a,i} = \sigma(b\#i) + w_{b,i}$, we see that $x_{a,c,i} = x_{b,c,i}$ and $x_{a,d,i} = x_{b,d,i}$. We rename these slack variables by removing the first port name from their subscripts to $x_{c,i}$, $x_{d,i}$ and so on. Now we see that again each variable from the equalities is associated with an edge from the derived precedence graph, the w variables correspond to the dotted edges, while the x variables correspond to the normal edges. With the extra constraints mentioned above, we formulate a schedule for the comparator component, which is given in Equation 3.11.

$$\begin{aligned}
\sigma(a\#i) &= \sigma(a\#0) + (\sum j : 0 \leq j < i : w_{c,j} + x_{a,j} + w_{a,j} + x_{c,j}) \\
\sigma(b\#i) &= \sigma(b\#0) + (\sum j : 0 \leq j < i : w_{c,j} + x_{b,j} + w_{b,j} + x_{c,j}) \\
\sigma(c\#i) &= \sigma(a\#0) + (\sum j : 0 \leq j < i : w_{c,j} + x_{a,j} + w_{a,j} + x_{c,j}) + w_{a,i} + x_{c,i} \\
\sigma(d\#i) &= \sigma(a\#0) + (\sum j : 0 \leq j < i : w_{d,j} + x_{a,j} + w_{a,j} + x_{d,j}) + w_{a,i} + x_{d,i}
\end{aligned} \tag{3.11}$$

Note that since $\sigma(b\#0) + w_{b,0} = \sigma(a\#0) + w_{a,0}$, we can substitute $\sigma(a\#0) + w_{a,0} - w_{b,0}$ for $\sigma(b\#0)$ in the schedule above. In general the schedule is completely determined by the time slot of one event plus the values of all the slack variables.

3.4.1 Component schedules

The previous section contains some examples of how to generate the equalities based on the schedule of a component. This section generalizes this idea and describes a method to construct these equalities by looking at the structure of components. At first, we do not look at the transformation of concurrent events as described in the previous section.

We show that it is fairly easy to generate a set of equalities for a component. We do this by first generating the precedence graph for a component and then transform the precedence graph into a set of equalities. This last step is fairly easy because the precedence graph can be mapped to a unique set of inequalities, which we transform to equalities by simply introducing slack variables with a positive value. If we were to look at the derived precedence graph of a component, we would introduce some slack variables with a positive value for the

normal edges and some other slack variables with a non-negative value for the terminated distribution edges.

Note that these precedence graphs are used in this case as an intermediate representation. It might not be necessary to construct or store it as a whole to use it to generate ILPs.

We are going to define some functions later on defined by using structural induction, so we start by introducing these structures.

- We have atomic statements, which are the communication statements like `port?var` or `port!expression`.
- If S_1 and S_2 are statements, then so is $S_1; S_2$.
- If S_1 and S_2 are statements, then so is S_1, S_2 .
- If E is an expression that has an integer value greater or equal than 0 and S is a statement, then so is $\#E[S]$.
- If S is a statement, then so is $\#[S]$ (the infinite repetition of S)

As previously stated, we want to construct a finite initial part of the precedence graphs. We specify such an initial part by stating the maximum number of events on each port or channel that we want to consider. To that end, we need to be able to ‘count’ the number of events and we introduce two functions, λ and δ , both with signature $\mathcal{A} \rightarrow \mathbb{N}$, to help us with that. For a port or channel a , the number $\delta(a)$ denotes the number of remaining events associated with a and similarly, the number $\lambda(a)$ denotes the number of completed events associated with a . We have that $\delta(a) + \lambda(a)$ is constant per port or channel a . We construct a finite downward closed subgraph by choosing initial values $\delta(a)$ for all ports and channels.

Furthermore, we have that if λ is the count directly preceding a statement $STAT$, then $\lambda[STAT]$ denotes the count after execution of statement $STAT$ and similarly for $\delta[STAT]$.

We use structural induction to define λ and δ .

$$\begin{aligned}
\delta[a?x](b) &= \begin{cases} (\delta(b) - 1) \uparrow 0 & , \text{ if } a = b \\ \delta(b) & , \text{ if } a \neq b \end{cases} & \lambda[a?x](b) &= \begin{cases} \lambda(b) + 1 & , \text{ if } a = b \\ \lambda(b) & , \text{ if } a \neq b \end{cases} \\
\delta[a!x](b) &= \begin{cases} (\delta(b) - 1) \uparrow 0 & , \text{ if } a = b \\ \delta(b) & , \text{ if } a \neq b \end{cases} & \lambda[a!x](b) &= \begin{cases} \lambda(b) + 1 & , \text{ if } a = b \\ \lambda(b) & , \text{ if } a \neq b \end{cases} \\
\delta[S_0 ; S_1](a) &= (\delta[S_0][S_1])(a) & \lambda[S_0 ; S_1](a) &= (\lambda[S_0][S_1])(a) \\
\delta[S_0 , S_1](a) &= (\delta[S_0][S_1])(a) & \lambda[S_0 , S_1](a) &= (\lambda[S_0][S_1])(a) \\
\delta[\#E[S]](a) &= \begin{cases} (\delta[S][\#(E-1)[S]])(a) & , \text{ if } E > 0 \\ \delta(a) & , \text{ if } E = 0 \end{cases} & \lambda[\#E[S]](a) &= \begin{cases} (\lambda[S][\#(E-1)[S]])(a) & , \text{ if } E > 0 \\ \lambda(a) & , \text{ if } E = 0 \end{cases} \\
\delta[\#[S]](a) &= \begin{cases} 0 & , \text{ if } a \in \mathcal{A}[S] \\ \delta(a) & , \text{ if } a \notin \mathcal{A}[S] \end{cases} & \lambda[\#[S]](a) &= \begin{cases} \infty & , \text{ if } a \in \mathcal{A}[S] \\ \lambda(a) & , \text{ if } a \notin \mathcal{A}[S] \end{cases}
\end{aligned} \tag{3.12}$$

We need to be able to talk about the set of first events (Equation 3.13) of a statement and the set of last events (Equation 3.14) of a statement. We do this using λ and δ as defined above.

$$\begin{aligned}
first(a?x) &= \begin{cases} \{a\#(\lambda(a))\} & , \text{ if } \delta(a) > 0 \\ \emptyset & , \text{ if } \delta(a) = 0 \end{cases} \\
first(a!x) &= \begin{cases} \{a\#(\lambda(a))\} & , \text{ if } \delta(a) > 0 \\ \emptyset & , \text{ if } \delta(a) = 0 \end{cases} \\
first(S_1; STAT) &= \begin{cases} first(S_1) & \text{ if } first(S_1) \neq \emptyset \\ first(STAT) & \text{ if } first(S_1) = \emptyset \end{cases} \\
first(S_1, STAT) &= first(S_1) \cup first(STAT) \\
first(\#E[S]) &= \begin{cases} \emptyset & \text{ if } E = 0 \\ first(S) & \text{ if } E > 0 \end{cases} \\
first(\#[S]) &= first(S)
\end{aligned} \tag{3.13}$$

$$\begin{aligned}
last(a?x) &= \begin{cases} \{a\#(\lambda(a))\} & , \text{ if } \delta(a) > 0 \\ \emptyset & , \text{ if } \delta(a) = 0 \end{cases} \\
last(a!x) &= \begin{cases} \{a\#(\lambda(a))\} & , \text{ if } \delta(a) > 0 \\ \emptyset & , \text{ if } \delta(a) = 0 \end{cases} \\
last(S_1; STAT) &= \begin{cases} last(STAT)[S_1] & \text{ if } last(STAT) \neq \emptyset \\ last(S_1) & \text{ if } last(STAT) = \emptyset \end{cases} \\
last(S_1, STAT) &= last(S_1) \cup last(STAT) \\
last(\#E[S]) &= \begin{cases} \emptyset & \text{ if } E = 0 \\ last(S) & \text{ if } E > 0 \end{cases} \\
last(\#[S]) &= \emptyset
\end{aligned} \tag{3.14}$$

With these definitions, we can use the structure of statements to generate the precedence graph of a component. We know that nodes of the precedence graph of component C are the events of component C , $\mathcal{E}(C)$. We define a function g that calculates for a statement, the edges in the precedence graph, corresponding to that statement.

$$\begin{aligned}
g(S_1; STAT) &= \{(e, f) | e \in last(S_1) \wedge f \in first(STAT)[S_1]\} \\
&\quad \cup g(S_1) \\
&\quad \cup g(STAT)[S_1] \\
g(S_1, STAT) &= \emptyset \\
g(\#E[S]) &= \begin{cases} \{(e, f) | e \in last(S) \wedge f \in first(S)[S]\} \\ \quad \cup g(S) \\ \quad \cup g(\#E - 1[S])[S] \end{cases} & \text{ if } E > 0 \\
&\quad \emptyset & \text{ if } E = 0 \\
g(\#[S]) &= \{(e, f) | e \in last(S) \wedge f \in first(S)[S]\} \\
&\quad \cup g(\#[S])[S]
\end{aligned} \tag{3.15}$$

We note that in the first equality from the equation above, we make use of $last(S_1)$, which is only defined if S_1 is not an infinite repetition, c.f. [13].

For component C with command $\#[S]$, we have Equation 3.16.

$$G(C) = (\mathcal{E}(C), g(S)) \tag{3.16}$$

Note that the precedence graph is a graph with an infinite number of nodes and edges because of the infinite repetition in a component. At this point, however, we do not care and we explain why in Section 4.1. For now, it suffices to say that we look at a downward closed subgraph of the precedence graph that contains nodes and edges for an initial number of blocks only.

With the precedence graph, we also implicitly have the inequalities, namely $\{\sigma(e\#i) < \sigma(f\#j) \mid (e\#i, f\#j) \in G.edges\}$. These inequalities are simply rewritten by introducing a non-negative slack variable. For inequality $\sigma(e\#i) < \sigma(f\#j)$ we introduce slack variable $x_{e,f,i\downarrow j}$, rewriting it to equality $\sigma(f\#j) = \sigma(e\#i) + x_{e,f,i\downarrow j}$.

We demonstrate this method for the $Split_3^1$ component by calculating the precedence graph, which is shown in Figure 3.5. We derive, using the precedence graph, the set of inequalities from Equation 3.17, we add slack variables and as a result we have the set of equalities from Equation 3.18.

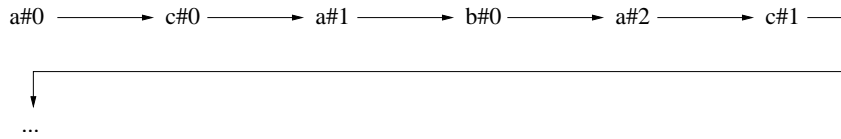


Figure 3.5: Precedence graph for the $Split_3^1$ component.

$$\begin{array}{ll}
 \sigma(a\#0) < \sigma(c\#0) & \sigma(b\#0) < \sigma(a\#2) \\
 \sigma(c\#0) < \sigma(a\#1) & \sigma(a\#2) < \sigma(c\#1) \\
 \sigma(a\#1) < \sigma(b\#0) &
 \end{array} \tag{3.17}$$

$$\begin{array}{ll}
 \sigma(c\#0) = \sigma(a\#0) + x_{a,c,0} & \sigma(a\#2) = \sigma(b\#0) + x_{b,a,0} \\
 \sigma(a\#1) = \sigma(c\#0) + x_{c,a,0} & \sigma(c\#1) = \sigma(a\#2) + x_{a,c,1} \\
 \sigma(b\#0) = \sigma(a\#1) + x_{a,b,0} &
 \end{array} \tag{3.18}$$

Similarly, we compute the precedence graph of the comparator component $G(Comparator)$, presented earlier in Figure 3.3. Note that this precedence graph is the ‘normal’ precedence graph, without the special edges denoting the w -slack variables that can be 0.

We generate the set of inequalities from this graph like we did before and we add slack variables to rewrite those inequalities to equalities. If we take a closer look at those equalities (Equation 3.8), we see that both $\sigma(c\#i)$ and $\sigma(d\#i)$ occur twice as a left-hand side in the set of equalities. This means that we cannot pick just any pair of values for the slack variables introduced in those equalities. This means that we are constrained in choosing values for the slack variables.

If we take a closer look at the nodes $c\#i$ and $d\#i$ from the precedence graph of the comparator component, we see that they have multiple incoming edges. We conclude that the nodes with multiple incoming edges in the precedence graph, constrain the possible values of the slack variables introduced for the equalities derived from that node.

With this remark in the back of our heads, we take another look at the precedence graph we constructed in Figure 3.4. If we count the number of nodes in that graph with multiple

incoming edges, we see that that number is only half the number of nodes in the ‘normal’ precedence graph with multiple incoming edges for the same number of blocks handled.

This means that it is possible to reduce the number of constraints on the values of the slack variables at the cost of a bit of uniformity in the precedence graph. In Chapter 5 on linear programming problems, we discuss why this might be beneficial.

3.4.2 System schedules

We have a notion of matching schedules for two subsystems that are connected via a channel. For systems X and Y , such that $\sigma_X(X) \in \Sigma(X)$ and $\sigma_Y(Y) \in \Sigma(Y)$ we define a predicate matching, denoted $\sigma_X \bowtie \sigma_Y$:

$$\sigma_X \bowtie \sigma_Y \equiv (\forall c, i : 0 \leq i \wedge c \in \mathcal{CS}(X\|Y) : \sigma_X(c\#i) = \sigma_Y(c\#i)) \quad (3.19)$$

As before, we use system structure to calculate schedules for systems. Since we have a method to calculate the set of inequalities concerning events of a system, given a precedence graph (Section 3.4.1), it would be nice if we could calculate the precedence graph of a system, because we would get the inequalities (and therefore the equalities) for free.

Luckily for us, calculating the precedence graph of a system $S = X\|Y$ is very straightforward. The nodes of the precedence graph are, as with components, $\mathcal{E}(S)$. For the edges, we simply define $g(S) = g(X) \cup g(Y)$. This means that we take the precedence graphs of subsystems X and Y and replace the formal parameters in the labels of the nodes with the actual parameters.

For example, we look at the precedence graph of a linear buffer with capacity 2 that has an input port ‘a’ and an output port ‘b’. The system is constructed by combining two buffer components via a channel ‘c’.

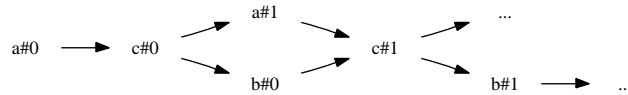


Figure 3.6: Precedence graph for a system consisting of 2 one-place buffers, connected via channel ‘c’.

Notice, that we introduced extra constraints by combining these systems. Nodes $c\#i$ for $i > 0$ now have multiple incoming edges. But this makes perfect sense, since we are dealing with components that communicate with each other via synchronous channels and synchronous communication means that both systems at the end of a channel are somewhat constrained in their behavior, they both have to be ready to communicate before communication takes place.

We remark again that there is the possibility of name clashes. In Section 2.2.1 we hinted at how these name clashes can be avoided.

Chapter 4

System performance

In Section 1.2.1 we have introduced some performance metrics. In this chapter we formalize these metrics and show how they are related.

4.1 Performance metrics

This section formalizes the performance metrics from Section 1.2.1. These formalizations are taken from [7]. We will see that some of the performance metrics are dependent on a schedule for a system.

4.1.1 Capacity

The capacity of a system is defined as the number of its storage locations. The capacity of a single component is given by the number of variables in the program text.

The capacity of a system X , denoted $\kappa(X)$, is then given as the sum of the capacities of the composed subsystems. Formally, for a system $X = Y_0 \parallel \dots \parallel Y_{n-1}$, we define:

$$\kappa(X) = (\sum i : 0 \leq i < n : \kappa(Y_i)) \tag{4.1}$$

4.1.2 I/o-distance

The i/o-distance $\delta(X)$ of a system X , is the average number of storage locations that a token visits on its passage through the system. To compute the i/o-distance, it is therefore sufficient to know the number of storage locations visited by each individual token. The latter quantity is not always known, unless we know the value of the token. However, for a system performing a block computation, this is calculated by counting the total number of storage locations that the tokens from one block visit and dividing that number by the block size of the system.

4.1.3 Cycle time and throughput

The cycle time of a system is a measure that says something about the average time between successive events. Its inverse is throughput, which says something about the average number of events per time slot.

To begin with, we define the notion of the j -th *individual cycle time at port a* as the number of time slots that pass between successive events $a\#j$ and $a\#(j+1)$. This is denoted

$\gamma_{a,j}$ and for a schedule σ , we have that

$$\gamma_{a,j}(\sigma) = \sigma(a\#(j+1)) - \sigma(a\#j) \quad (4.2)$$

Using the individual cycle time at a port a , we calculate the average cycle time at that port for the initial i successive events by calculating the sum of the first i individual cycle times and dividing by the number of cycles i :

$$\frac{(\sum_{j: 0 \leq j < i} \gamma_{a,j}(\sigma))}{i} \quad (4.3)$$

Because of the definition of $\gamma_{a,j}$, this is equal to:

$$\frac{\sigma(a\#i) - \sigma(a\#0)}{i} \quad (4.4)$$

If we take the limit for i approaching infinity for this quantity, we get the *average cycle time* $\Gamma_a(\sigma)$ at port $a \in \mathcal{PS}(X)$ of system X . This means, it is given by:

$$\Gamma_a(\sigma) = \lim_{i \rightarrow \infty} \frac{\sigma(a\#i)}{i} \quad (4.5)$$

If we know that the average time between consecutive events on a port a is $\Gamma_a(\sigma)$, then we also know that the average rate at which events take place at that port is $\frac{1}{\Gamma_a(\sigma)}$. This means that the *average throughput at port a* of a system X , denoted $\Theta_a(\sigma)$, is the inverse of the average cycle time at that port, i.e.,

$$\Theta_a(\sigma) \times \Gamma_a(\sigma) = 1 \quad (4.6)$$

We call a system *token (or data) conservative* if no tokens are ‘lost’ or ‘created’ in the system. Stated otherwise, the number of input actions and the number of output actions of the system as a whole are equal. This means that for token conservative systems, the sum of the average throughputs at the input ports has to be equal to the sum of the average throughputs at the output ports, i.e.,

$$(\sum_{p: p \in \mathcal{PS}_{in}(X)} \Theta_p(\sigma)) = (\sum_{q: q \in \mathcal{PS}_{out}(X)} \Theta_q(\sigma)) \quad (4.7)$$

This is the case because if the average rate at which input actions take place differs from the average rate at which output actions take place, the system is either accepting more tokens from or delivering more tokens to its environment. **SPSs** are finite systems, which means the number of tokens that can be present in the system is bounded from below by 0 and bounded from above by the capacity of the system κ . If the system is accepting more tokens than it is delivering, it means that at some point tokens have to be destroyed, contradicting data conservativeness. On the other hand, if the system is delivering more tokens than it is accepting, it means that at some point tokens have to be created, again contradicting data conservativeness. So this proves that data conservative systems have to have equal average throughput at the input ports and the output ports.

The notion of the average throughput of a system only makes sense for token conservative systems. The systems considered in this thesis all have this nice property. We define the average throughput of a token conservative system as the sum of either the average throughputs on all the input ports or all the output ports:

$$\Theta(\sigma) = (\sum_{p: p \in \mathcal{PS}_{in}(X)} \Theta_p(\sigma)) = (\sum_{q: q \in \mathcal{PS}_{out}(X)} \Theta_q(\sigma)) \quad (4.8)$$

Given the average throughput of a system, we define the *average cycle time*, denoted Γ , as its inverse.

$$\Gamma(\sigma) \times \Theta(\sigma) = 1 \quad (4.9)$$

It is important to note that the metrics presented in this section depend on a schedule σ .

When we look at single input, single output systems, like block sorter systems, we see that the average throughput is bound from below by 0 and bound from above by $\frac{1}{2}$. This upper bound arises because the component on the input side and the component on the output side both have a maximum throughput of $\frac{1}{2}$, thus bounding the maximum throughput of the system.

4.1.4 Latency and occupancy

Next, we look at latency, which is a measure that says something about how much time it takes for a token to traverse the system on average. This thesis investigates block computations, which means that all the values in output block i , only depend on values from input block i , and we use this property to define the notion of *block latency*. It is defined by the sum of the time slots of the output events of the block minus the sum of the time slots of the input events of the block, divided by the block size. For instance, for a system with input port a and output port b , executing schedule σ , the block latency $\lambda_p(\sigma)$ is given by

$$\lambda_p(\sigma) = \frac{1}{n} (\sum_{j: 0 \leq j < n: \sigma(b\#(np+j))} - \sigma(a\#(np+j))) \quad (4.10)$$

Using the block latency, we calculate the average latency for the first q blocks:

$$\Lambda^{(q)}(\sigma) = \frac{(\sum_{p: 0 \leq p < q: \lambda_p(\sigma)})}{q} \quad (4.11)$$

Taking the limit for q to infinity of $\Lambda^{(q)}(\sigma)$, we get the *average latency*, denoted $\Lambda(\sigma)$, i.e.,

$$\Lambda(\sigma) = \lim_{q \rightarrow \infty} \Lambda^{(q)}(\sigma) = \lim_{q \rightarrow \infty} \frac{1}{q} (\sum_{p: 0 \leq p < q: \lambda_p(\sigma)}) \quad (4.12)$$

Another measure we are interested in for our performance analysis is the average number of tokens present in the system: the occupancy of a system. We define the *instantaneous occupancy* at time slot j , denoted $\omega_j(\sigma)$, as the number of tokens present in the system at time slot j . Note that the instantaneous occupancy of system X is always an integer number between 0 and $\kappa(X)$. We calculate instantaneous occupancy by taking the number of tokens (or data-items) that have entered the system before time slot j and subtracting the number of tokens that have left the system before time slot j :

$$\begin{aligned} \omega_j(\sigma) &= (\sum_{a: a \in \mathcal{PS}_{in}(X): \#\{i|\sigma(a\#i) < j\}}) \\ &- (\sum_{b: b \in \mathcal{PS}_{out}(X): \#\{i|\sigma(b\#i) < j\}}) \end{aligned} \quad (4.13)$$

We calculate the average occupancy from time slot 0 to a certain time slot, say t by calculating the instantaneous occupancy at all time slots $0 \leq j < t$ and dividing that number by t :

$$\frac{(\sum_{j: 0 \leq j < t: \omega_j(\sigma)})}{t} \quad (4.14)$$

If we take the limit of this quantity, we get the *average occupancy*, denoted by $\Omega(\sigma)$, so it is defined by:

$$\Omega(\sigma) = \lim_{t \rightarrow \infty} \frac{1}{t} (\sum j : 0 \leq j < t : \omega_j(\sigma)) \quad (4.15)$$

We note that since the instantaneous occupancy lies between 0 and $\kappa(X)$, it follows that the average occupancy also lies between 0 and $\kappa(X)$.

If we know the average rate at which data-items enter and leave the system and we know how long each data-item remains in the system on average, then we can also compute the average number of data-items in the system. This means that average occupancy, average throughput and average latency are related, which is captured in *Little's Law* [5]:

$$\Omega(\sigma) = \Theta(\sigma) \times \Lambda(\sigma) \quad (4.16)$$

Since average cycle time is the inverse of average throughput, this is equal to:

$$\Lambda(\sigma) = \Gamma(\sigma) \times \Omega(\sigma) \quad (4.17)$$

We have the notions of minimal and maximal average occupancy. They are calculated by taking a fixed average throughput $\Theta = \theta$ and varying over all possible schedules σ of a system that achieve this fixed average throughput:

$$\Omega \downarrow_{\Theta=\theta} (X) = \downarrow \{ \Omega(\sigma) \mid \Theta(\sigma) = \theta \} \quad (4.18)$$

Similarly, we have the notions of minimal and maximal average latency. Furthermore, since cycle time and throughput are related, we can substitute throughput with cycle time. This means that we have the following variations:

$$\begin{aligned} \Omega \downarrow_{\Theta=\theta} (X) &= \downarrow \{ \Omega(\sigma) \mid \Theta(\sigma) = \theta \} & \Omega \downarrow_{\Gamma=\gamma} (X) &= \downarrow \{ \Omega(\sigma) \mid \Gamma(\sigma) = \gamma \} \\ \Omega \uparrow_{\Theta=\theta} (X) &= \uparrow \{ \Omega(\sigma) \mid \Theta(\sigma) = \theta \} & \Omega \uparrow_{\Gamma=\gamma} (X) &= \uparrow \{ \Omega(\sigma) \mid \Gamma(\sigma) = \gamma \} \\ \Lambda \downarrow_{\Theta=\theta} (X) &= \downarrow \{ \Lambda(\sigma) \mid \Theta(\sigma) = \theta \} & \Lambda \downarrow_{\Gamma=\gamma} (X) &= \downarrow \{ \Lambda(\sigma) \mid \Gamma(\sigma) = \gamma \} \\ \Lambda \uparrow_{\Theta=\theta} (X) &= \uparrow \{ \Lambda(\sigma) \mid \Theta(\sigma) = \theta \} & \Lambda \uparrow_{\Gamma=\gamma} (X) &= \uparrow \{ \Lambda(\sigma) \mid \Gamma(\sigma) = \gamma \} \end{aligned} \quad (4.19)$$

Note that because of the definition of average latency (Equation 4.12), this means amongst others that:

$$\Lambda \uparrow_{\Theta=\theta} (X) = \uparrow \{ \lim_{q \rightarrow \infty} \Lambda^{(q)}(\sigma) \mid \Theta(\sigma) = \theta \}$$

The values for the performance metrics described in this section are bounded. We have already shown that $0 \leq \Omega \leq \kappa(X)$ and, by definition, we have that $\Omega \downarrow_{\Theta=\theta} \leq \Omega \uparrow_{\Theta=\theta}$, meaning that we have the following bounds:

$$0 \leq \Omega \downarrow_{\Theta=\theta} (X) \leq \Omega \uparrow_{\Theta=\theta} (X) \leq \kappa(X) \quad (4.20)$$

We show in Section 7.1.1 that we can further refine these bounds.

Finally we remark that the metrics described in this section depend on some schedule σ , but in general we do not care exactly what that schedule is, as long as the average throughput (or consequently average cycle time) is fixed.

4.1.5 Elasticity

Although systems run at an average throughput, this does not mean that the actual throughput at any give time is equal to this average throughput. There are fluctuations in the throughput and we call this throughput jitter. The elasticity of a system is a measure of how well the system can deal with throughput jitter at a given average throughput. To calculate the elasticity $\varepsilon_\theta(X)$ of a system X for a certain fixed average throughput $\Theta = \theta$, we calculate the maximal occupancy for that throughput, subtract the minimal occupancy for that throughput and divide the result by the capacity $\kappa(X)$ of system X . We remark that both minimal and maximal average occupancy are between 0 and the $\kappa(X)$ of system X , which means that $0 \leq \varepsilon(X) \leq 1$.

Formally, the elasticity of system X at average throughput θ is defined by:

$$\varepsilon_\theta(X) = \frac{\Omega \uparrow_{\Theta=\theta}(X) - \Omega \downarrow_{\Theta=\theta}(X)}{\kappa(X)} \quad (4.21)$$

In this thesis, we plot graphics similar to that of Figure 4.1. This means that we read the elasticity indirectly from these figures that plot normalized occupancies as a function of throughput.

Figure 4.1 shows the minimal and maximal occupancies at given throughputs for both a bubble sort system and an insertion sort system with block size 2 calculated for 1 block. The elasticity at a certain throughput can be derived from this figure by calculating the difference between the left and right flank at that throughput. Figures like this one, make it easy to visually compare the elasticity of systems. For instance, we easily see that the insertion sort system can achieve higher maximal throughput and that it is more elastic at any given throughput than the bubble sort system. Moreover, we see that the insertion sort system can achieve higher maximal occupancy than the bubble sort system at (very) low throughput.

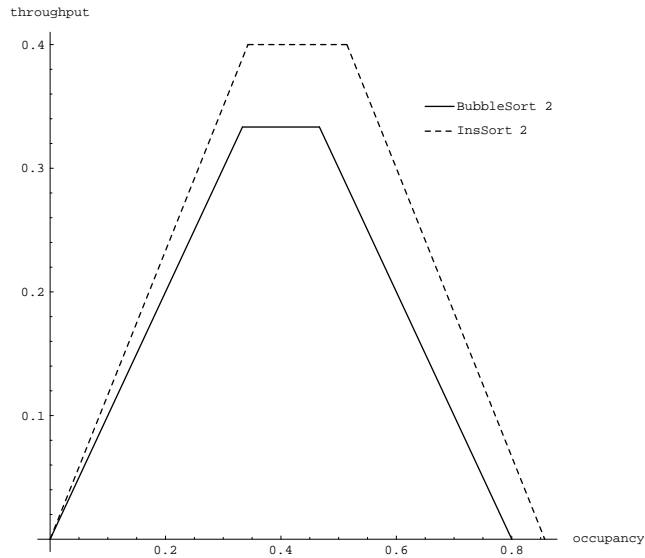


Figure 4.1: Example elasticity plot for block sorters with block size 2 for 1 block.

Chapter 5

Integer linear programming problems

In this Chapter, we show that it is possible to translate the problem of finding the extreme (discrete time) schedules for a given system into integer programming problems and how to translate the solutions of these programming problems back to the actual schedules. We introduce integer linear programming (ILP) problems and we show that these ILP problems can be constructed in a compositional manner by using the structure of systems. We construct ILP problems by giving small ILP problems for the basic components and a method to combine ILP problems of instances of subsystems to form the ILP of a bigger system.

An integer linear programming (ILP) problem is an optimization problem. A solution for the problem is a set of integer values for a number of decision variables x_i , such that the value of the (linear) ‘objective function’ is minimal (or maximal) and all (linear) constraints are satisfied.

If we look at the following ILP problem:

- Objective function to be minimized, OF :

$$(\sum_{i: 1 \leq i \leq n} c_i x_i) .$$

- Constraints on the decision variables:

$$\begin{aligned} (\sum_{i: 1 \leq i \leq n} a_{1,i} x_i) &= b_1 \\ (\sum_{i: 1 \leq i \leq n} a_{2,i} x_i) &= b_2 \\ &\vdots \\ (\sum_{i: 1 \leq i \leq n} a_{m,i} x_i) &= b_m \end{aligned}$$

- Bounds for the decision variables

$$\begin{aligned} x_1 &\geq d_1 \\ &\vdots \\ x_n &\geq d_n \end{aligned}$$

We can use a compact notation, found in linear algebra, to define these ILPs. We express these problems in matrix form (note that we use all vectors as column vectors, that is $n \times 1$

matrices). Added benefit is that it is easy to manipulate matrices and vectors. We introduce vectors \mathbf{c} and \mathbf{x} that hold the coefficients c_i of the objective function and the decision variables x_i respectively, matrix A and vector \mathbf{b} that hold the constraint coefficients and the constraint values b_i respectively and vector \mathbf{d} that holds the bounds for the variables d_i .

Formally, a ILP problem is defined by the tuple $(\mathbf{x}, \mathbf{c}, A, \mathbf{b}, \mathbf{d})$ for which we have:

$$\begin{aligned}
 \text{Minimize} \quad & \mathbf{c}^T \mathbf{x} \\
 \text{Subject to} \quad & \mathbf{x} \in \mathbb{Z}^n \\
 & A\mathbf{x} = \mathbf{b} \\
 & \mathbf{x} \geq \mathbf{d}
 \end{aligned} \tag{5.1}$$

The revised simplex algorithm [9] can be used to solve these ILPs. The worst-case complexity of this algorithm is exponential, however, it has a very reasonable average-case complexity. Furthermore, the number of constraints of the ILP seems to be a very important factor in the average-case behavior of the algorithm. There are other (better) algorithms to solve programming problems, however, Mathematica, the application that we use to solve the programming problems uses the revised simplex method to solve larger integer variants of the programming problems.

As we have shown in Section 3.4.1, we can reduce the number of constraints in concurrent components, at the cost of losing some uniformity for the slack variables. A reduced number of constraints might shorten the time needed to calculate the schedules if we use an existing implementation of the revised simplex algorithm. However, if we were to implement our own revised simplex algorithm, we could perhaps take advantage of the uniformity of the lower bound of the slack variables.

5.1 ILP problems of individual components

For component X with individual block size N , we define the objective function $OF_n(X)$ to be the total latency of component X after the first n blocks of N values have been processed. This means that we take the sum of the time slots of the output events of n blocks and we subtract the sum of the time slots of the input events of those blocks.

We first look at the one-place buffer component, which has individual block size 1. The objective function we are looking for is the latency of the component handling a number of blocks, which, for data conservative systems, is defined as the time slots of all output events of those blocks minus the time slots of all input events of those blocks. For a one-place buffer, the time slot of the first output event is defined by $\sigma(b\#0) = \sigma(a\#0) + x_{a,b,0}$, the time slot of the first input event is defined by $\sigma(a\#0)$. This means that the latency for the first block of values for a one place buffer component, $OF_1(Buf)$, is defined by $\sigma(b\#0) - \sigma(a\#0) = x_{a,b,0}$.

In this thesis we look at programming problems with fixed average throughput (and accordingly fixed cycle time) where the average is taken over a number of blocks. For example, by looking at a one-place buffer running at average throughput $\theta = \frac{1}{3}$, we see that if the first input event takes place at time slot t_0 , the second input event has to take place at time slot $t_0 + 1/\theta = t_0 + 3$ for the average throughput of that one block to be $\frac{1}{3}$. (We have made use of the periodic behavior of the component.) Since we showed previously that $\sigma(a\#1) = \sigma(a\#0) + x_{a,b,0} + x_{b,a,0}$, which consequently means in this case that $x_{a,b,0} + x_{b,a,0} = 3$.

We define for a system X running at average throughput θ and handling n number of blocks, the function $ILP(X, \theta, n)$ that is the integer linear programming problem.

For example, the programming problem for a one-place buffer component running at average throughput θ and handling one block is given by:

$$ILP(Buf, \theta, 1) = \left(\begin{pmatrix} x_{a,b,0} \\ x_{b,a,0} \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, (1 \ 1), \left(\frac{1}{\theta} \right), \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right) \quad (5.2)$$

We use notation $\mathbf{1}_n$ to denote a column-vector of dimension n that contains all 1's.

The next example we look at, are *Split* components. By design, a *Split* 1_3 component has block size 3. The time slots of the input events of the first block are $\sigma(a\#0)$, $\sigma(a\#1)$ and $\sigma(a\#2)$, while the time slots of the output events of the first block are given by $\sigma(b\#0) = \sigma(a\#1) + x_{a,b,0}$, $\sigma(c\#0) = \sigma(a\#0) + x_{a,c,0}$ and $\sigma(c\#1) = \sigma(a\#2) + x_{a,c,1}$. This means that the total latency (and also our objective function) for the *Split* 1_3 component handling a single block is $x_{a,c,0} + x_{a,b,0} + x_{a,c,1}$.

This is calculated by taking the sum over the time slots at which the output events take place and subtracting from this the sum over the time slots at which the input events take place. For the *Split* 1_3 component, we derived the schedule and the corresponding set of equalities. We apply some transformations to show that we can generate an integer programming problem for a component by using its schedule and other structural properties.

The first step we take is that we construct a vector \mathbf{x} of variable names, it contains the names of the slack variables that appear in the schedule. These variables are the decision variables for the integer programming problems. In the case of this instance of the split component, \mathbf{x} would be $(x_{a,c,0}, x_{c,a,0}, x_{a,b,0}, x_{b,a,0}, x_{a,c,1}, x_{c,a,1})^T$. Note that we added slack variable $x_{c,a,1}$ to vector \mathbf{x} , although it does not occur in the schedule of the first block. The reason we do this is explained later on, but for now it is enough to know that that variable will play a role in the constraints of the ILP. We rewrite the equalities of the *Split* 1_3 component in such a way that all right-hand sides of the equalities are expressed in terms of one constant value ($\sigma(a\#0)$) plus a number of slack variables. Note that the slack variables we introduce, can be mapped directly to the edges in the precedence graph of this *Split* component.

$$\begin{aligned} \sigma(a\#0) &= \sigma(a\#0) &&= \sigma(a\#0) + (0, 0, 0, 0, 0, 0) \mathbf{x} \\ \sigma(c\#0) &= \sigma(a\#0) + x_{a,c,0} &&= \sigma(a\#0) + (1, 0, 0, 0, 0, 0) \mathbf{x} \\ \sigma(a\#1) &= \sigma(a\#0) + x_{a,c,0} + x_{c,a,0} &&= \sigma(a\#0) + (1, 1, 0, 0, 0, 0) \mathbf{x} \\ \sigma(b\#0) &= \sigma(a\#0) + x_{a,c,0} + x_{c,a,0} + x_{a,b,0} &&= \sigma(a\#0) + (1, 1, 1, 0, 0, 0) \mathbf{x} \\ \sigma(a\#2) &= \sigma(a\#0) + x_{a,c,0} + x_{c,a,0} + x_{a,b,0} + x_{b,a,0} &&= \sigma(a\#0) + (1, 1, 1, 1, 0, 0) \mathbf{x} \\ \sigma(c\#1) &= \sigma(a\#0) + x_{a,c,0} + x_{c,a,0} + x_{a,b,0} + x_{b,a,0} + x_{a,c,1} &&= \sigma(a\#0) + (1, 1, 1, 1, 1, 0) \mathbf{x} \end{aligned}$$

Next, we combine this into matrix form, which results in:

$$\begin{pmatrix} \sigma(a\#0) \\ \sigma(c\#0) \\ \sigma(a\#1) \\ \sigma(b\#0) \\ \sigma(a\#2) \\ \sigma(c\#1) \end{pmatrix} = \begin{pmatrix} \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \mathbf{x}$$

We call the vector on the left-hand side of this equation \mathbf{e} which is the vector of time slots of the events of the component. We call the matrix on the right-hand side of this equation M which is the matrix containing the coefficients belonging to the slack variables.

The objective function is the latency induced by the component and it is calculated (for data-conservative components) by taking the sum of the time slots of the output events and subtracting from that value the sum of the time slots of the input events of the component. Since it is possible to calculate the value of all the time slots for a single component by adding a constant to a number of slack variables, we see that the total latency for a data-conservative component is expressed only in terms of slack variables.

For this specific component, this means that it is $\sigma(c\#0) + \sigma(c\#1) + \sigma(b\#0) - \sigma(a\#0) - \sigma(a\#1) - \sigma(a\#2)$, which we can rewrite to $(-1, 1, -1, 1, -1, 1) \mathbf{e}$ and we derive:

$$\begin{aligned}
& (-1, 1, -1, 1, -1, 1) \mathbf{e} \\
= & \\
& (-1, 1, -1, 1, -1, 1) \begin{pmatrix} \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \\ \sigma(a\#0) \end{pmatrix} + (-1, 1, -1, 1, -1, 1) \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \mathbf{x} \\
= & \\
& (1, 0, 1, 0, 1, 0) \mathbf{x}
\end{aligned}$$

We show how to construct vector $\mathbf{e}_{io} = (-1, 1, -1, 1, -1, 1)$ by looking at structural properties. With this vector, we can calculate vector \mathbf{c}^T that holds the coefficients of the slack variables of the objective function of the ILP. We get \mathbf{c}^T by multiplying \mathbf{e}_{io} with matrix M that we construct from the schedules.

We define for each port $p \in \mathcal{PS}(X)$ of a component X , a vector \mathbf{e}_p of size $\dim(\mathbf{e})$ that consists of 0's and 1's in such a way that $\mathbf{e}_p(i) = 1$ if and only if the port associated with event $\mathbf{e}(i)$ is equal to p :

$$(\forall i : 0 \leq i < \dim(\mathbf{e}_p) : \mathbf{e}_p(i) = [\text{port}(\mathbf{e}(i)) = p])$$

For example, for the $Split_3^1$ component we have $\mathbf{e}_a = (1, 0, 1, 0, 1, 0)^T$, $\mathbf{e}_b = (0, 0, 0, 1, 0, 0)^T$ and $\mathbf{e}_c = (0, 1, 0, 0, 0, 1)^T$. Remark that since all events take place on a port of the component, we have that $(\sum p : p \in \mathcal{PS}(X) : \mathbf{e}_p) = \mathbf{1}_{\dim(\mathbf{e})}$.

The vector \mathbf{e}_{io} that we are looking for to construct \mathbf{c}^T from the ILP, is computed by taking the sum of all the \mathbf{e}_p vectors for output ports p and subtracting the sum of all the \mathbf{e}_q vectors for input ports q .

$$\mathbf{e}_{io} = \left(\left(\sum_{p \in \mathcal{PS}_{out}(X)} \mathbf{e}_p \right) - \left(\sum_{p \in \mathcal{PS}_{in}(X)} \mathbf{e}_p \right) \right)^T$$

This means that we have that:

$$\mathbf{c}^T = \mathbf{e}_{io} \times M$$

So far we have given vector \mathbf{x} and vector \mathbf{c} of the ILP of the $Split_3^1$ component. The decision variables introduced all have a lower bound of 1, so we have that vector $\mathbf{d} = \mathbf{1}_6$. The only thing remaining are the constraints of the ILP, described by matrix A and vector \mathbf{b} . The only constraint we have for the $Split_3^1$ component running at throughput θ , is that the first event of the second block should take place $3/\theta$ time slots later than the first event of the

first block. We know that the first event of the second block is $\sigma(a\#3) = \sigma(a\#0) + x_{a,c,0} + x_{c,a,0} + x_{a,b,0} + x_{b,a,0} + x_{a,c,1} + x_{c,a,1}$, so we have as (only) constraint that $(\mathbf{1}_6)^T \mathbf{x} = 3/\theta$. This means that we have $A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$ and $\mathbf{b} = (3/\theta)$ and we define:

$$ILP(Split_3^1, \theta, 1) = \left(\begin{pmatrix} x_{a,c,0} \\ x_{c,a,0} \\ x_{a,b,0} \\ x_{b,a,0} \\ x_{a,c,1} \\ x_{c,a,1} \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, (1 \ 1 \ 1 \ 1 \ 1 \ 1), (3/\theta), \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \right) \quad (5.3)$$

We solve the integer programming problems for the $Split_3^1$ component running at $\theta = \frac{1}{3}$ as an example. The problem to solve: find values \mathbf{v} such that $\mathbf{c}^T \mathbf{v}$ is minimal, $\mathbf{v} \geq \mathbf{1}_6$ and $\mathbf{1}_6^T \mathbf{v} = 9$. We see that $\mathbf{c}^T \mathbf{x} = x_{a,c,0} + x_{a,b,0} + x_{a,c,1}$, and if we want to minimize this, we need to choose $x_{a,c,0}$, $x_{a,b,0}$ and $x_{a,c,1}$ as small as possible. The smallest value possible for each x is 1. Now we still have to choose values for $x_{c,a,0}$, $x_{b,a,0}$ and $x_{c,a,1}$. Since they do not occur in the objective function, it does not matter for the minimal value of the objective function what value they get. We are only constrained by the lower bounds and the constraint of the programming problem. We see that, with the values already chosen, the problem comes down to finding a solution for $x_{c,a,0} + x_{b,a,0} + x_{c,a,1} = 6$ with respect to the lower bounds of all x . We see that a solution that has a minimal value for the objective function is $\mathbf{v} = (1, 1, 1, 1, 1, 4)^T$, but it is not the only solution that has this minimal value, we could have just as well chosen $\mathbf{v} = (1, 4, 1, 1, 1, 1)^T$ as the solution. This means that different solutions exist that have a minimal value for the objective function.

If we would have chosen $\theta = 1$, than we would not be able to find a solution for the problem, because you would not be able to find values \mathbf{v} that would meet all the requirements. On the one hand, you would have to choose all x at least 1, on the other hand, the sum of all x would have to be less than 3. This means that it is possible that with a given throughput θ , an ILP has no solution, meaning that there is no valid schedule that respects the ordering of all the events and achieves throughput θ . On the other hand, if there is a throughput θ for which the ILP has a minimal solution, then that solution is minimal for all values for the throughput smaller than θ .

Maximizing the objective function is equal to minimizing the inverse of the objective function, which in this case means minimizing $-x_{a,c,0} - x_{a,b,0} - x_{a,c,1}$. We see that the larger we choose the values of these three variables, the more minimal the solution of the objective function gets. The minimal value of the objective function is reached when we choose $x_{a,c,0} + x_{a,b,0} + x_{a,c,1} = 6$ and hence $x_{c,a,0} = 1$, $x_{a,b,0} = 1$ and $x_{c,a,1} = 1$. Again, there are several values that we can choose for the remaining three variables that all give the maximal solution for the objective function, e.g. $\mathbf{v} = (4, 1, 1, 1, 1, 1)^T$ and $\mathbf{v} = (2, 1, 2, 1, 2, 1)^T$. Again we note that with a fixed throughput, there might not be a solution that achieves maximal latency.

Contrary to the case of minimal latency, there is no solution that is, in general, maximal for all values for the throughput. If we choose a smaller throughput, it is possible that we find a larger maximal value of the objective function.

We take another look at the one-place buffer. We derive that for a one-place buffer handling one block, $\mathbf{e}_a = (1, 0)^T$, $\mathbf{e}_b = (0, 1)^T$, which leads to $(-1, 1)$. Furthermore we have

$$M = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}.$$

What happens if we look at that buffer handling two blocks? We note that vector \mathbf{e}_a for the second block is exactly the same as that for the first block. This also holds for vector \mathbf{e}_b . This means that if we combine these two vectors, we end up with the same result as for the first block. This makes constructing the vector for a component handling n blocks very easy. We use for vectors v and w of sizes n and m respectively, notation $(v, w) = (v_0, \dots, v_{n-1}, w_0, \dots, w_{m-1})$. Now we simply have that for all ports p :

$$\mathbf{e}_p^n = (e_p^{n-1}, e_p) \quad (5.4)$$

If it is possible to use a similar construction for matrix M , then we can easily compute the ILP for a component handling n blocks. First we note that we included a slack variable in \mathbf{x} that was introduced as the difference between the last event of the first block and the first event of the second block. We used that slack variable before in the constraints of the ILP. Again we note that for the second block matrix M is equal to matrix M for the first block.

We define matrix $0_{n \times m}$ to be the $n \times m$ -matrix filled with all 0's. Furthermore, we use the following notation, for matrices A , B , C and D of respective sizes $i \times j$, $l \times j$, $i \times k$ and $l \times k$ we can construct matrix E of size $(i + l) \times (j + k)$:

$$E = \begin{pmatrix} A & C \\ B & D \end{pmatrix}$$

With the matrix M of the component handling one block, that has size $p \times q$, and matrix Q_r the matrix that has r rows with the coefficients of the slack variables of the first event of the second block, we define $M^1 = M$ and M^n as follows:

$$M^n = \begin{pmatrix} M & 0_{p,q(n-1)} \\ Q_{p(n-1)} & M^{n-1} \end{pmatrix} \quad (5.5)$$

We look at the one-place buffer handling two blocks. For a less strict periodicity, we see that to obtain the previous average throughput of $\frac{1}{3}$, we only require that the third input event (which is the first event of the third block) has to take place at time slot $t_0 + 2/\theta = t_0 + 6$. We know that $\sigma(a\#3) = x_{a,b,0} + x_{b,a,0} + x_{a,b,1} + x_{b,a,1}$, meaning that this requires that $x_{a,b,0} + x_{b,a,0} + x_{a,b,1} + x_{b,a,1} = 6$. The programming problem for the one-place buffer component running at average throughput θ_{Buf} and handling two blocks is given by:

$$ILP(Buf, \theta, 2) = \left(\begin{pmatrix} x_{a,b,0} \\ x_{b,a,0} \\ x_{a,b,1} \\ x_{b,a,1} \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, (1 \ 1 \ 1 \ 1), \left(\frac{2}{\theta} \right), \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \right) \quad (5.6)$$

Thus far, we have only considered components without concurrent statements. We look at the *Comparator* component to demonstrate how we construct ILPs in the case that a component does have concurrent statements. We use the schedule from Equation 3.11 that we got from the derived precedence graph for the construction of the various components of the ILP. For one block we get the vector $\mathbf{x} = (w_{a,0}, w_{b,0}, x_{c,0}, x_{d,0}, w_{c,0}, w_{d,0}, x_{term})^T$ with the

decision variables. We have vector $\mathbf{e} = (a\#0, b\#0, c\#0, d\#0)^T$ with the events and for vector \mathbf{e}_{i_0} that we use in combination with matrix M to compute the objective function, the same technique is used. From the schedule we derive matrix M :

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

This means that we can calculate vector $\mathbf{c} = (3, -1, 1, 1)^T$ that describes the coefficients of the objective function of the ILP.

Note that we introduce 3 slack variables that are currently unused. These are two slack variables for the distributed termination of events $c\#0$ and $d\#0$ and one slack variable that has the same role as the extra slack variable for the other components, it is used for the constraints.

If we look at the bounds for the decision variables, we now have that they are not all equal to 1 since some of these variables are introduced for distributed termination, and we have that $\mathbf{d} = (0, 0, 1, 1, 0, 0, 1)^T$.

Because of the concurrency in the component, we now have extra constraints. If we look at the derived precedence graph for the comparator component, we see that there are multiple paths leading to the distributed termination nodes. Since a node is mapped to a single time slot, this means that the sums of the values of the slack variables on all paths with the same end-point, have to be equal. In this case, it means that $a\#0 + w_{a,0} = b\#0 + w_{b,0}$ and $a\#0 + w_{a,0} + x_{c,0} + w_{c,0} = a\#0 + w_{a,0} + x_{d,0} + w_{d,0}$. For each node that represents the distributed termination of n events, we get $n - 1$ equalities. We calculate vectors \mathbf{p} and \mathbf{q} and rewrite each equality to vector notation, meaning that we have $\mathbf{p}\mathbf{e}^T = \mathbf{q}\mathbf{e}^T$. We then append vector $\mathbf{p} - \mathbf{q}$ to matrix A that contains the left-hand sides of the constraints from the ILP and we append 0 to vector \mathbf{b} that contains the right hand sides of the constraints from the ILP.

This means that we constructed all the components of the ILP and we have the integer linear programming problem for the comparator component:

$$ILP(Comparator, \theta, 1) = \left(\begin{pmatrix} w_{a,0} \\ w_{b,0} \\ x_{c,0} \\ x_{d,0} \\ w_{c,0} \\ w_{d,0} \\ x_{term} \end{pmatrix}, \begin{pmatrix} 3 \\ -1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 & -1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \frac{2}{\theta} \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right) \quad (5.7)$$

5.2 Constructing programming problems of combined systems

Since communications we consider are synchronous, combining components in a system will put more constraints on the ordering of events. After all, two components have to be ready to commence a communication action.

In a system, instances of subsystems are linked via channels. If we look at the ordering of the events of a channel, we see that there are constraints imposed on the ordering for both components that are connected to this channel.

In the previous section, we took the precedence graph of a component, constructed the schedule from that graph and used the schedule plus some structural properties of the component to calculate the ILP.

We combine the parts of the ILPs of the various subsystems that we construct with the method described in the previous section, to construct the ILP of the system as a whole. We demonstrate this with a simple example.

We consider a system consisting of two buffer components that are connected via a channel c , e.g. $Buf(a, c) || Buf(c, b)$. The input of the system is port a and the output is port b . We can construct the local matrices M for both components and we can combine them into a matrix M for the whole system by using the technique described in Equation 5.5. The first event of the buffer on the output side of the system is on channel c and we can use the event on channel c from the buffer on the input side of the system to combine these matrices. First we note that the names of the slack variables have thus far been chosen arbitrarily and from now on we just denote the vector containing the decision variables \mathbf{x} .

We combine $M_{Buf1} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ with $M_{Buf2} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ and we get matrix

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

For vector \mathbf{e}_{io} that we combine with matrix M to compute the objective function, we use the same technique described before. We combine the vectors \mathbf{e} of both components and we get the vector $\mathbf{e} = (a\#0, c\#0, c\#0, b\#0)^T$ for the system as a whole. We construct \mathbf{e}_{io} by replacing the input events with -1 , the output events with 1 and the events on internal channels with 0 . This means we get $\mathbf{e}_{io} = (-1, 0, 0, 1)^T$ and the resulting vector $\mathbf{c} = (1, 0, 1, 0)^T$ is the vector with the coefficients for the slack variables of the objective function.

As for matrix A and vector \mathbf{b} , we can combine these easily, since all constraints introduced thus far are local constraints. However, by combining the components, we introduce extra constraints. For each channel, we have that all the events on that channel have to be scheduled at the same time slot in both components. However, these constraints can be added easily, since matrix M contains two rows for each event on a channel, once for the component on the input side, once for the component on the output side. Since these rows have to denote the same value, we can subtract these rows and append the resulting row to the matrix A of constraints and we append 0 to the vector \mathbf{b} that contains the right hand sides of these constraints.

This means that we have combined the different parts of the ILP from two components to form the parts of the ILP of the system as a whole. For the system of the example, we have that:

$$ILP(Buf(a, c) || Buf(c, b), \theta, 1) = (\mathbf{x}, \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} \frac{2}{\theta} \\ \frac{2}{\theta} \\ 0 \end{pmatrix}, \mathbf{1}_4)$$

5.3 ILP solver

We use an off the shelf mathematical application called *Mathematica* [4] to solve the ILPs that we generate.

Chapter 6

Tool

The tool written to aid in constructing ILPs can be viewed as a compiler. The input language is the programming language used to describe these systems and the compiled output is a set of ILPs in a format that can be imported directly in *Mathematica*. Any compiler can be seen as being split in two parts. The first part consists of a parser that parses program text, checks some context conditions and generates an attributed parse tree. The parser and the resulting attributed parse tree are described in more detail in Section 6.1. The second part is the code-generation, which uses the information in the parse tree to generate code in an output format, in our case, the set of ILPs. This is described in more detail in Section 6.2.

The tool is programmed in an object oriented fashion in the Python programming language [3].

6.1 Parser

The parsing step of the compiler is split up in smaller tasks. The first step in parsing input is done by a scanner. Typically, a scanner splits the input into smaller pieces that have a semantical meaning, called tokens. A scanner, for instance, distinguishes *identifiers* from *numbers*.

We have implemented the scanner by making use of regular expression pattern matching. This means that we have given regular expressions for all the tokens that we have and try to match the longest possible substring, starting at the first character of the first string of input, against these patterns. If a match is found for a token, then the type of token, line number and column are stored in the token list and the matching part is removed from the input string. In the erroneous case that the start of the string does not match the pattern for any token, an error message is shown to the user, detailing the location in the input string where the error occurred.

We remark that our implementation of the parser starts with scanning the whole input at once. Other implementations of parsers combine the steps of scanning and parsing, the parser then simply requests the next token from the scanner when it is needed.

When the scanning step is completed, the parser simultaneously:

- checks the syntax (Section 6.1.1)
- verifies context conditions (Section 6.1.2)
- constructs an attributed parse tree (Section 6.1.3)

6.1.1 Checking the syntax

We have programmed a recursive descent parser that accepts the language described by the attribute grammar of Appendix A. This is done by using the methods described in great detail in [12]. This implementation uses an object oriented approach for our recursive descent parser.

6.1.2 Verifying context conditions

The grammar of the language we are using, which is given in Appendix A, has several context conditions. These conditions are checked in the functions of the recursive descent parser described in the previous section, as described in [12].

6.1.3 Constructing the attributed parse tree

Again, we used the methods described in [12] to generate an attributed parse tree.

6.2 Generating the ILP from a parse tree

First, we use the attributes generated in the previous step to flatten the parse tree. The values of the parameters from the system declarations are known at this stage and we use them to substitute system instances with their according component instances. Furthermore, we use the values for the parameters to unfold the repetitions, with the exception of the infinite repetition, in the body of these components. Simultaneously with the flattening of the parse tree, we use a global counter to enumerate all the channels of the system. When a channel is declared in the program text, it is mapped to a unique number and in the body of the system channel names are substituted with their unique number. Moreover, we are only interested at the communication behavior at this point in the process, which means that we ignore the other aspects of the components, like the names of the variables. Since the components that we use in this thesis have empty initializations, we have not implemented methods that deal components that have non-empty initializations and this is left as future research.

This process results in a single list with the communication behavior of the components that are defined with unique channel numbers.

The tool computes the block size of the component on the input side of the system, which in most cases is the block size for the system as a whole. However, the user is asked to verify this by entering the block size of the system. Moreover, the user is asked to enter the number of blocks to generate the ILP for. With the block size of the system and the number of blocks to compute, we can calculate for all ports and channels the number of events taking place at that port or channel. Combining this with the communication behavior results in a downward closed subgraph of the precedence graph. Nodes for the distributed termination of concurrent events (described in Section 3.3) are added to the graph and the resulting derived precedence graph is used to generate the various vectors and matrices that represent the ILPs.

The user is asked to enter the name of an output file. The vectors and matrices that are constructed by the compiler are written to this file in such a way that the file can be imported in and evaluated by Mathematica. Furthermore, variables that contain for instance the block size, the number of blocks computed and the capacity of the system are also added to this file. We use Mathematica to open this file and evaluate it. The result is that the functions to

compute the values for the decision variables that achieve minimal and maximal total latency are loaded into memory. These functions have the total cycle time as a parameter.

Chapter 7

Results

The goal of this project is to analyze and compare the elasticity of block sorter designs. One of those designs is the family of bubble sorters. An analysis based on the structure of these bubble block sorters, in particular an analysis of the minimal and maximal latencies at a given average cycle time, is presented in [7]. We use the results from this analysis as a benchmark for the tool presented in this thesis, meaning that the first experiment uses our tool to determine minimal and maximal latencies at given average cycle times. Then we compare the resulting values for minimal and maximal latency to the values of minimal and maximal latency that we find using the structure analysis.

We start by describing the family of bubble block sorter systems and then we give a summary of the results from the structure analysis given in [7]. This is followed by the general setup of the experiments conducted and subsequently we describe the experiments that we have conducted and discuss the results.

7.1 Bubble block sorter systems

Bubble block sorters that sort blocks of size n use bubble cells called $Bubble_n$. For every $n \geq 2$, bubble cell $Bubble_n$ is a small system consisting of four basic components and it is a single-input-single-output stream processing system that performs a block computation. Figure 7.1 provides the program text and corresponding diagram.

From an operational point of view, it can be seen that when every input block of $Bubble_n$ has a sorted tail of certain length l , the output block of $Bubble_n$ has a sorted tail of length $l + 1$. This means that a bubble block sorter that sorts blocks of size n , denoted BBS_n , has to consist of $n - 1$ of these bubble cells $Bubble_n$, or more formal:

$$\begin{aligned} BBS_2 &= Bubble_2 \\ BBS_{n+1} &= (\|_{2 \leq j \leq n+1} Bubble_{n+1}) \end{aligned} \tag{7.1}$$

The *Split*, *Merge* and *FMLS* components each contribute 1 to the capacity of the bubble cell and the *Comparator* component contributes 2 to the bubble cell, which means that $\kappa(Bubble_n) = 5$. For the capacity of a bubble block sorter system composed of $n - 1$ of these $bubble_n$ components, we derive that $\kappa(BBS_n) = 5n - 5$.

Next, we look at the average i/o-distance of a bubble cell $bubble_n$. Since communications do not depend on the values of the data-items, the average i/o-distance of a system is the same no matter how we choose the values of the data-items. Consider a block of data-items such

```

BubbleCell = syst(in a & out b & val n) .
|[ chan c0, c1, c2, c3, c4, c5
[>
  Split(a, c0, c1, n, 0)    ||
  FMLS(c0, c2, c3, c4, n)  ||
  Comparator(c1, c3, c2, c5) ||
  Merge(c4, c5, b, n, n-1)
]|

```

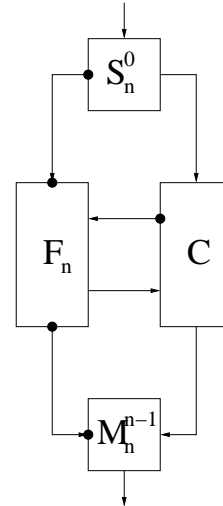


Figure 7.1: Bubble cell of order n .

that the first value is the maximum of the block and the remaining values are in ascending order. The first value gets passed from the *Split* to the *FMLS* component, which passes it to the *Comparator* component. The first data-item has thus far passed 3 components and since it has the largest value of all data-items in its block, it is passed $n - 1$ times to the *FMLS* component, once for each remaining $n - 1$ data-items in the block. The *FMLS* component passes this data-item a total of $n - 1$ times to the *Comparator* component, before finally passing it to the *Merge* component. This leads to a total i/o-distance of $2n + 1$ for the first data-item. The other $n - 1$ data-items are passed from the *Split* component to the *Comparator* where they are compared to the first (larger) data-item, meaning they get passed to the *Merge* component directly. Therefore, these data-items pass 3 components on their way through the system, thus the total distance traveled by all data-items of one block is $(2n + 1) + 3(n - 1) = 5n - 2$, leading to average i/o-distance $\frac{5n-2}{n}$.

7.1.1 Performance of bubble block sorter systems

In Section 4.1.4 we have stated that both the instantaneous and the average occupancy of a system X lie between 0 and the capacity of that system, $\kappa(X)$, which means we have trivial bounds for system X :

$$0 \leq \Omega(X) \leq \kappa(X)$$

If we take other properties of systems into account, for instance the average i/o-distance $\delta(X)$ of system X , it is possible to further refine these bounds. Recall that the average i/o-distance of a system is the average number of storage locations a token passes in the system. Of course, ‘passing a storage location’ means that a component in the system reads the value on an input port to a variable and then writes the value of that variable on an output port. These actions cannot take place in the same timeslot, so we know that this induces a latency of at least one time slot. Combining these facts, we know that the average latency of system X is bound from below by its i/o-distance:

$$\delta(X) \leq \Lambda(X)$$

We also give an upper bound for the average latency. We note that this upper bound is dependant on the average cycle time since with a higher cycle time, outputs can possibly be delayed, increasing the latency. Each component of a system X can at most induce an average maximal latency of the average cycle time, $\Gamma(X)$, and there are $\kappa(X)$ components in the system. However, we have to subtract the i/o-distance from this number, for the same reason that we added the i/o-distance to the lower bound: reading and writing of the same variable can not happen in the same timeslot. This means that we have an upper bound for the average latency for a system X running at average cycle time $\Gamma(X)$:

$$\Lambda(X) \leq \Gamma(X) \times \kappa(X) - \delta(X)$$

Consequently, because of Little's Law, which states that the average latency of a system is equal to the product of the average cycle time and the average occupancy of that system, this also imposes bounds on the occupancy of system X :

$$\Theta(X) \times \delta(X) \leq \Omega(X) \leq \kappa(X) - \Theta(X) \times \delta(X) \quad (7.2)$$

When the average throughput approaches 0, we see that we get the trivial bounds for the average occupancy:

$$0 \leq \Omega(X) \leq \kappa(X) \quad (7.3)$$

Note that all the bounds introduced so far are general bounds that hold for all systems. It is possible that the functionality of a system imposes synchronization constraints on the events of that system. These synchronization constraints can further restrict the set of possible schedules for a system. This means that it is possible that the average occupancy does not reach the lower or upper bounds.

Research in [7] further refines these bounds for bubble block sorter systems by taking the structure of these systems into account. This research shows that the minimal average latency for bubble cells is equal to the average i/o-distance of a bubble-cell, which is 5. Furthermore it shows that the maximal average latency for a bubble cell of order n running at average cycle time γ is given by $4\gamma - \frac{3n+4}{n}$.

$$\begin{aligned} \Lambda \downarrow_{\Gamma=\gamma} (Bubble_n) &= 5 \\ \Lambda \uparrow_{\Gamma=\gamma} (Bubble_n) &= 4\gamma - \frac{3n+4}{n} \end{aligned} \quad (7.4)$$

If we write this in terms of average occupancies and use that average throughput is the inverse of average cycle time, we get:

$$\begin{aligned} \Omega \downarrow_{\Theta=\theta} (Bubble_n) &= 5\theta \\ \Omega \uparrow_{\Theta=\theta} (Bubble_n) &= 4 - \theta \times \frac{3n+4}{n} \end{aligned} \quad (7.5)$$

Since a bubble block sorter system of order n is a composition of $n - 1$ of these bubble cells, we know that $\Omega \downarrow (BBS_n)$ cannot be less than $(n - 1) \times \Omega \downarrow (Bubble_n)$. This is of course similar to the maximal average occupancy of such a system, which means that we have more accurate bounds for bubble block sorter systems.

$$\begin{aligned} \Omega \downarrow_{\Theta=\theta} (BBS_n) &\geq (n - 1)(5\theta) \\ \Omega \uparrow_{\Theta=\theta} (BBS_n) &\leq (n - 1) \left(4 - \theta \times \frac{3n+4}{n} \right) \end{aligned} \quad (7.6)$$

Analytically speaking, we cannot easily deduce whether there exist schedules for which the occupancy assumes this minimal or maximal value. Initially, we thought that these schedules existed and that these inequalities were actually equalities. However, further experiments with the tool we have created show that this is not the case for all bubble block sorter systems.

7.2 General approach

The experiments we are going to conduct are focussed on elasticity. Recall that the elasticity of a system is calculated by subtracting the minimal occupancy of a system from the maximal occupancy of that system and then normalizing the result by dividing by the capacity of the system, which is the same as first normalizing the minimal and maximal occupancies and then subtracting them.

Results are presented in figures similar to that of Figure 4.1, which plot the average throughput as a function of the normalized occupancy. These figures make it easy to say something about minimal and maximal normalized occupancy (by looking at the left and right flanks respectively), the elasticity (by looking at the difference between these flanks) and the maximal throughput.

With a fixed average cycle time, and thus a fixed average throughput, we use Equation 4.16 to transform the minimal and maximal latencies to minimal and maximal occupancies. If we divide these values by the capacity of the system, we get the minimal and maximal normalized occupancy. The difference between these values is the elasticity for the system at cycle time $\Gamma = \gamma$. In terms of the units presented earlier, we have:

$$\frac{\Omega}{\kappa} = \frac{\Lambda}{\kappa\gamma} \quad (7.7)$$

Using the tool presented in this thesis, we transform the description of a system combined with the block size and the number of blocks we want to calculate to an ILP of any number of blocks of input of that system and some other properties of the system. This is saved in a file that can be imported in Mathematica. Amongst others, the file contains:

- variable *Blocks* which is an input parameter for the tool and holds the number of blocks for the ILP
- variable *BlockSize* which is equal to the block size of the system
- variable *Capacity* that holds the total capacity κ of the system
- a vector *vecCF* that contains the coefficients of the decision variables for the programming problem that give minimal and maximal latency
- functions *lpmin(ct_{total})* and *lpmax(ct_{total})* that yield vectors of values for the decision variables that are the solution of the ILPs for *total* cycle time *ct_{total}*

We use the results of the latter functions to calculate the minimal and maximal *total* latency of the first *Blocks* blocks by multiplying them with *vecCF*. Furthermore, it is possible to calculate actual schedules that give minimal and maximal latency.

For example, we calculated the ILPs for *BBS₂* handling one block of values and then we used Mathematica to solve *lpmin(6)* and *lpmax(6)* to get the schedules from Figure 7.2 that

result in minimal total latency 10 and maximal total latency 14 for one block of 2 values. This means that we have calculated for average cycle time $\Gamma = \frac{6}{2}$, the average minimal latency $\Lambda^{(1)} \downarrow_{\Gamma=3} = 5$ and average maximal latency $\Lambda^{(1)} \uparrow_{\Gamma=3} = 7$. This means that it is now possible to calculate the elasticity of the system at average cycle time 3:

$$\begin{aligned}
 \varepsilon_{\Gamma=3} &= \frac{\Omega \uparrow_{\Gamma=3} - \Omega \downarrow_{\Gamma=3}}{\kappa} \\
 &= \frac{\Lambda \uparrow_{\Gamma=3} - \Lambda \downarrow_{\Gamma=3}}{\kappa \Gamma} \\
 &= \frac{7 - 5}{5 * 3} \\
 &= \frac{2}{15}
 \end{aligned} \tag{7.8}$$

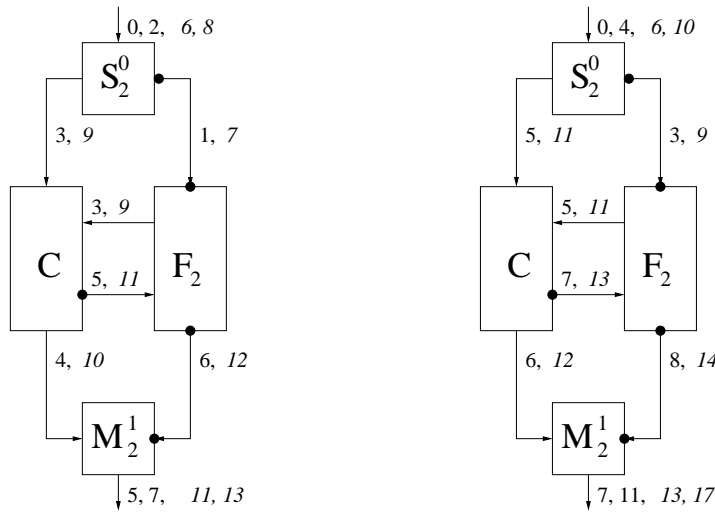


Figure 7.2: Schedules for minimal and maximal latencies of BBS_2 for average cycle time 3

7.3 Calculating actual values for performance metrics

The goal of the first set of experiments we conduct, is to find the real values of the performance metrics and to determine the number of blocks that we need to compute schedules for in order to determine these values. We construct ILPs for a system such that we calculate schedules with given fixed cycle time, say γ , that give minimal or maximal latency for a given (finite) number of initial blocks. If we calculate the schedules for a larger number (for instance an infinite number) of initial blocks, it is possible that we end up with more valid schedules with average cycle time γ . For example, if we calculate (repeating) schedules with an average cycle time 3 for a system with block size 2, we have:

| Blocks per cycle | Input (or output) events per cycle | Period of a cycle |
|------------------|------------------------------------|-------------------|
| 1 | 2 | 6 |
| 2 | 4 | 12 |
| ... | ... | ... |
| n | $2n$ | $6n$ |

The set of schedules calculated by considering a period of 1 block have an equal schedule in the set of schedules calculated by taking periods of 2, 3 or n blocks, but the contrary is not always true, e.g. for average cycle time 3, the schedule constructed by considering a period of 2 blocks that schedules the four events at time slots 5, 7, 9 and 11 does not have an equivalent schedule in the set of schedules calculated by considering a period of 1 block.

It is possible that these extra schedules achieve a lower minimal or higher maximal latency. Thus, by using the techniques from this thesis, we determine values that give an *approximation* of the actual values for the performance metrics. These approximations converge to the actual values since, by definition, the minimal (maximal) occupancy is monotonically decreasing (increasing) with respect to the number of blocks we consider. We expect that the rate of this convergence is high, meaning that a small number of blocks are enough to determine the actual values.

Furthermore, we assume that if we calculate minimal (or maximal) average latency for a larger number of blocks and it is equal to the minimal (or maximal) average latency of a smaller number of blocks, we have found the actual minimal (or maximal) latency. Formally, $\Lambda^{(i+1)} \uparrow = \Lambda^{(i)} \uparrow \implies \Lambda \uparrow = \Lambda^{(i)} \uparrow$.

7.3.1 Experiment description

We calculate for some bubble block sorter systems BBS_n and for a certain number of blocks i , the ILPs for the first i blocks of input with varying fixed average cycle time $\Gamma = \gamma$. We solve these programming problems, which give us for these i the minimal and maximal latencies: $\Lambda^{(i)} \downarrow_{\Gamma=\gamma}$ and $\Lambda^{(i)} \uparrow_{\Gamma=\gamma}$. Using this empirical data, we try to determine the limit of $\Lambda^{(q)} \uparrow_{\Gamma=\gamma}$ and $\Lambda^{(q)} \downarrow_{\Gamma=\gamma}$ for q going to infinity, which would give the actual average minimal and maximal latencies.

We conducted this experiment with the parameters from Table 7.1.

| System | Average cycle times | Number of blocks |
|---------|---|------------------|
| BBS_2 | $\frac{6}{2}, \frac{7}{2}, \dots, \frac{19}{2}, \frac{20}{2}$ | 1, 2, 3, 4, 5, 6 |
| BBS_3 | $\frac{8}{3}, \frac{9}{3}, \dots, \frac{21}{3}, \frac{22}{3}$ | 1, 2, 3, 4, 5, 6 |
| BBS_4 | $\frac{10}{4}, \frac{11}{4}, \dots, \frac{23}{4}, \frac{24}{4}$ | 1, 2, 3, 4, 5, 6 |
| BBS_5 | $\frac{12}{5}, \frac{13}{5}, \dots, \frac{25}{5}, \frac{26}{5}$ | 1, 2 |
| BBS_6 | $\frac{14}{6}, \frac{15}{6}, \dots, \frac{27}{6}, \frac{28}{6}$ | 1, 2 |
| BBS_7 | $\frac{16}{7}, \frac{17}{7}, \dots, \frac{29}{7}, \frac{30}{7}$ | 1, 2 |

Table 7.1: Parameters for the first set of experiments.

7.3.2 Experiment results

When we examine the results from this experiment, we see that for each system and each average cycle time from the table above, the values of $\Lambda^{(i)} \uparrow$ (and $\Lambda^{(i)} \downarrow$) are equal for all the number of blocks, i , that we have calculated.

This means that the extremal latencies we calculate from the ILPs of one block for these systems appear to be equal to the actual average latencies. So, we think that the schedules computed for one block of these bubble sorter systems achieve average minimal and maximal

latencies, or more formal:

$$\begin{aligned} \Lambda^{(1)} \downarrow_{\Gamma=\gamma} (BBS_n) &= \Lambda \downarrow_{\Gamma=\gamma} (BBS_n) \\ \Lambda^{(1)} \uparrow_{\Gamma=\gamma} (BBS_n) &= \Lambda \uparrow_{\Gamma=\gamma} (BBS_n) \end{aligned} \tag{7.9}$$

Although we only need to calculate one block to determine the average minimal and maximal occupancies, this does not mean that we are not interested in calculating more blocks of values anymore. For instance, if we would want to calculate the minimal or maximal average latency of the BBS_2 system at average cycle time $3\frac{1}{4}$, we would have to calculate a multiple of 2 blocks, because the total cycle time (which is the average cycle time multiplied by the block size multiplied by the number of blocks) that we provide as input for the programming problems needs to be an integer.

7.4 Comparing performance metrics with the bounds

The goal of this set of experiments is to compare the minimal and maximal occupancy at certain fixed cycle times with the bounds described in Section 7.1.1. To that end, we plot the values that we calculate from the ILPs and the values that we calculate from the analysis of the bounds in the same figure. We expect that the minimal and maximal occupancies coincide with the bounds given previously.

7.4.1 Experiment description

We calculate the normalized minimal and maximal occupancies for one block with the parameters from Table 7.2. Furthermore, we calculate the bounds for the minimal and maximal occupancies given in Section 7.1.1 in Equations 7.2 and 7.6.

| System | Average cycle times |
|---------|--|
| BBS_2 | $\frac{6}{2}, \frac{7}{2}, \dots, \frac{19}{2}, \frac{20}{2}$ and $\frac{10^p}{2}$ for $p = 2, \dots, 5$ |
| BBS_3 | $\frac{8}{3}, \frac{9}{3}, \dots, \frac{21}{3}, \frac{22}{3}$ and $\frac{10^p}{3}$ for $p = 2, \dots, 5$ |
| BBS_4 | $\frac{10}{4}, \frac{11}{4}, \dots, \frac{23}{4}, \frac{24}{4}$ and $\frac{10^p}{4}$ for $p = 2, \dots, 5$ |
| BBS_5 | $\frac{12}{5}, \frac{13}{5}, \dots, \frac{25}{5}, \frac{26}{5}$ and $\frac{10^p}{5}$ for $p = 2, \dots, 5$ |
| BBS_6 | $\frac{14}{6}, \frac{15}{6}, \dots, \frac{27}{6}, \frac{28}{6}$ and $\frac{10^p}{6}$ for $p = 2, \dots, 5$ |
| BBS_7 | $\frac{16}{7}, \frac{17}{7}, \dots, \frac{29}{7}, \frac{30}{7}$ and $\frac{10^p}{7}$ for $p = 2, \dots, 5$ |
| BBS_8 | $\frac{18}{8}, \frac{19}{8}, \dots, \frac{31}{8}, \frac{32}{8}$ and $\frac{10^p}{8}$ for $p = 2, \dots, 5$ |

Table 7.2: Parameters for the second set of experiments.

7.4.2 Experiment results

The calculated values for minimal and maximal occupancies are plotted along with the theoretical bounds for each bubble sorter system in Figures 7.3 to 7.9.

The first thing we note, is that the computed values for the minimal occupancy coincide with the bounds specific for the bubble sorter systems. However, this is not the case for the values of the maximal occupancy. More remarkable even, is that the computed values for maximal occupancy do not always fall on a straight line, for instance in Figure 7.5, there is a bend at point $(\frac{4}{7}, \frac{2}{7})$.

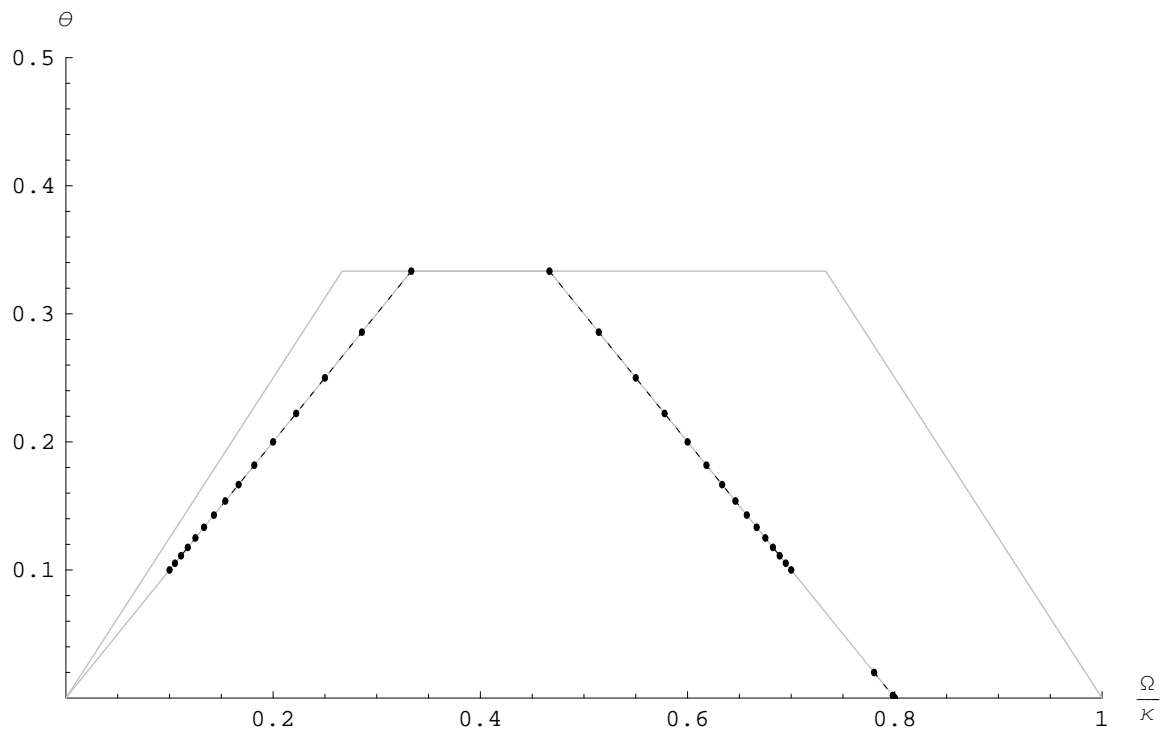


Figure 7.3: BBS_2

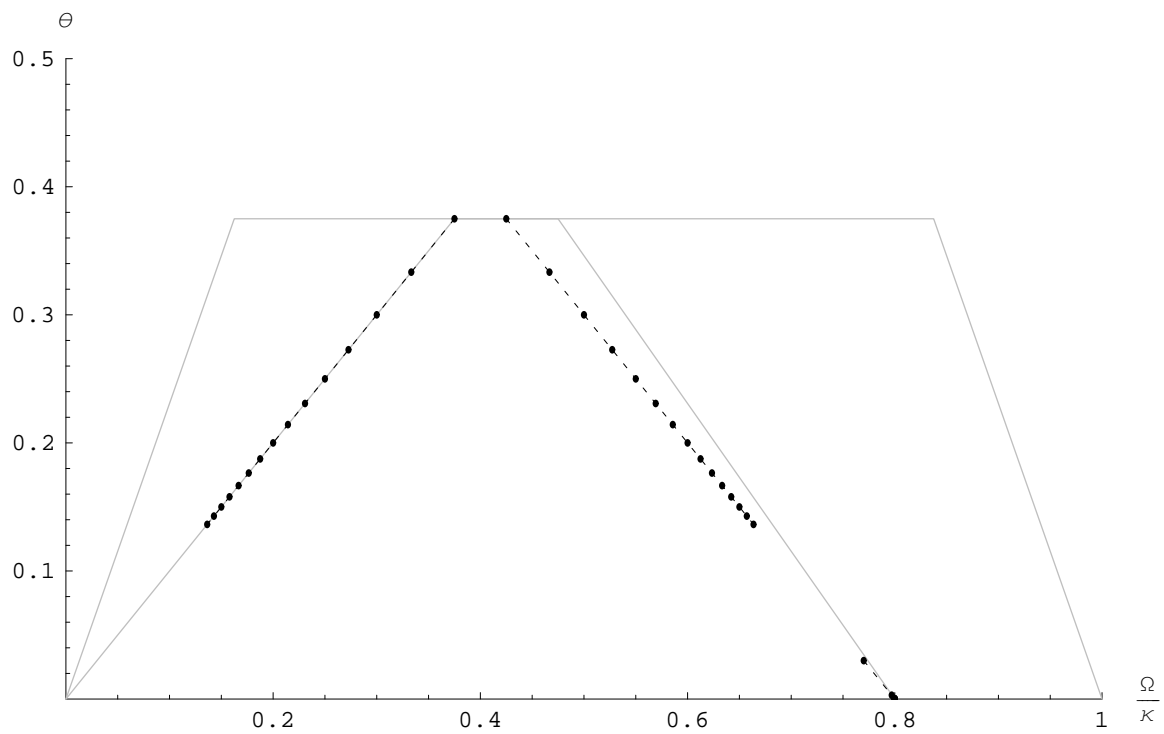


Figure 7.4: BBS_3

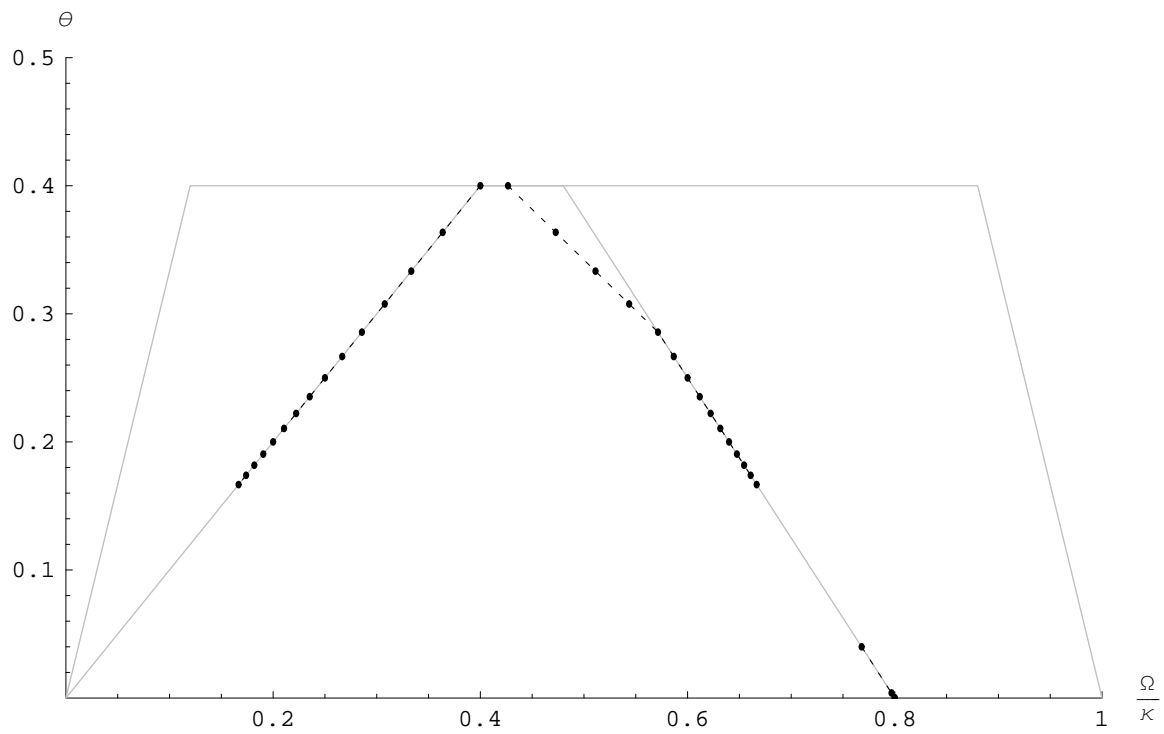


Figure 7.5: BBS_4

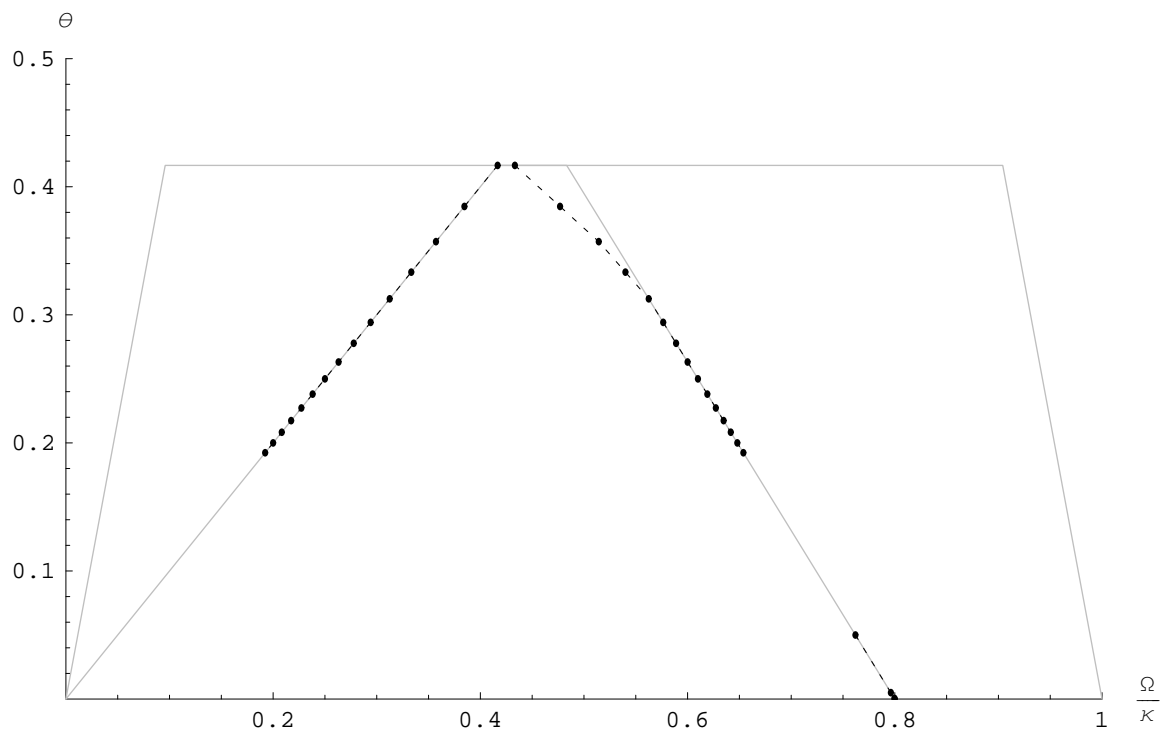


Figure 7.6: BBS_5

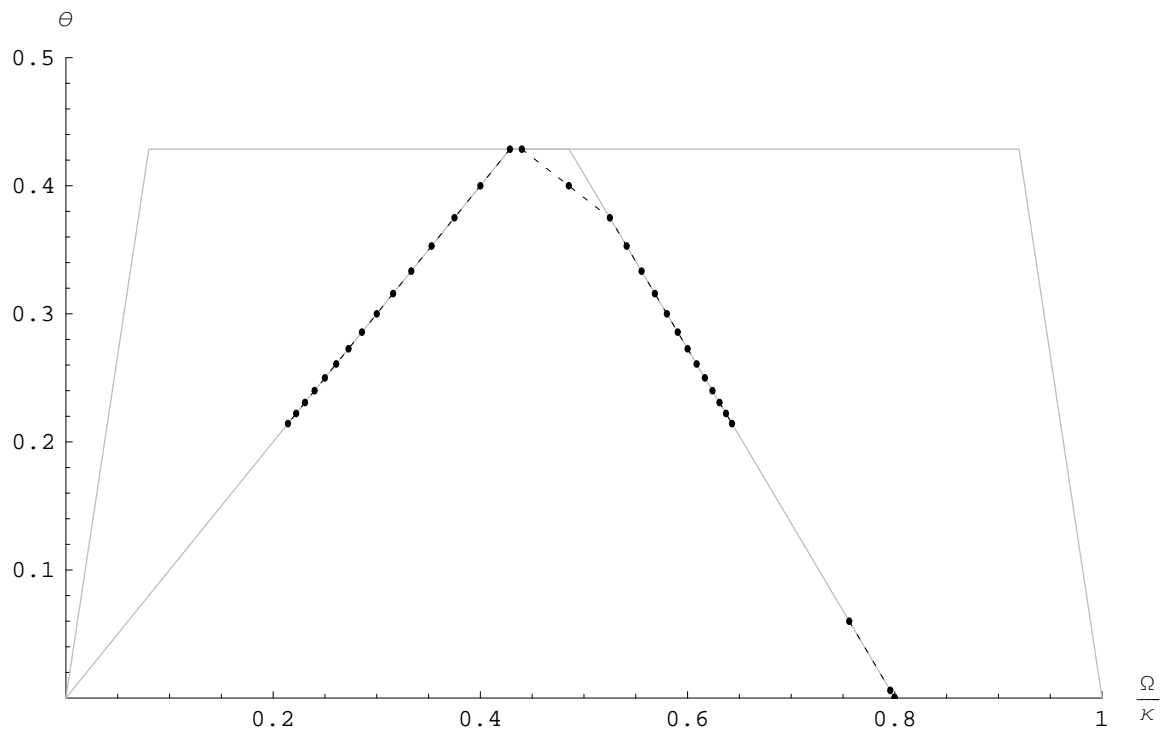


Figure 7.7: BBS_6

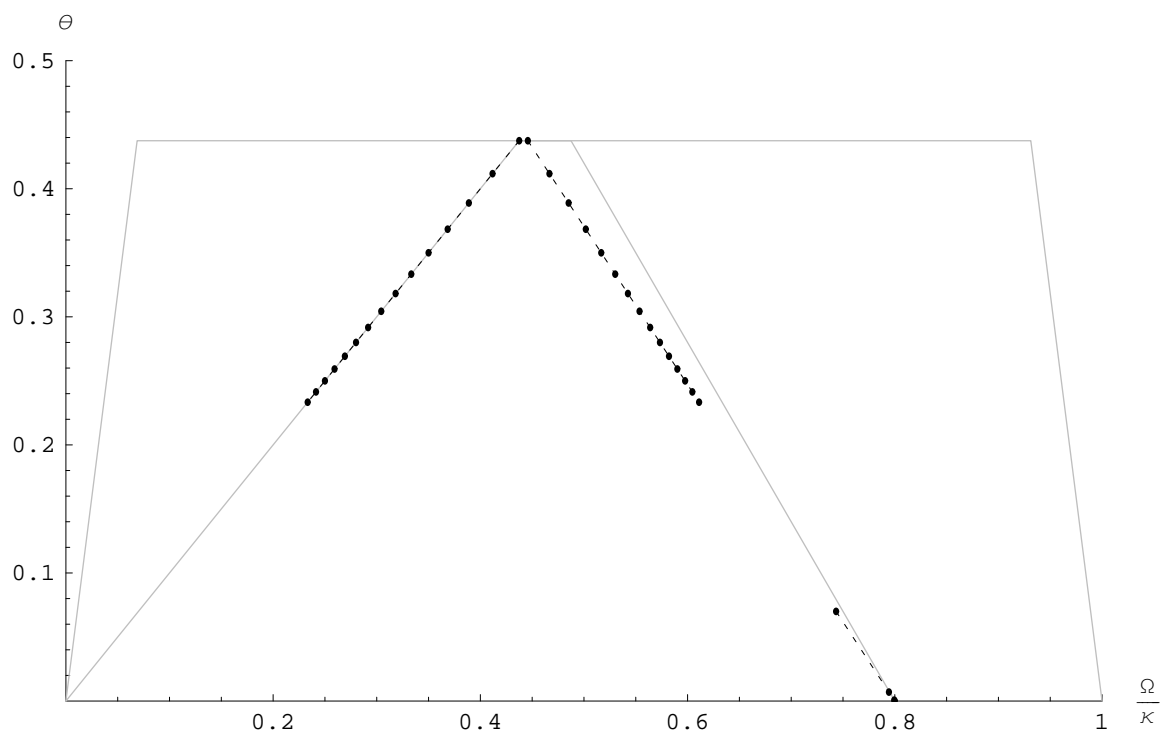


Figure 7.8: BBS_7

In general we see that for the experiments that we have conducted, there are three possible outcomes for the values of the maximal occupancy. They fall on a straight line that coincides with the bound (Figure 7.3), they fall on a straight line that does not coincide with the bound (Figures 7.4 and 7.8) or they do not fall on a straight line, but form several connected lines and one of these lines coincides with the bound.

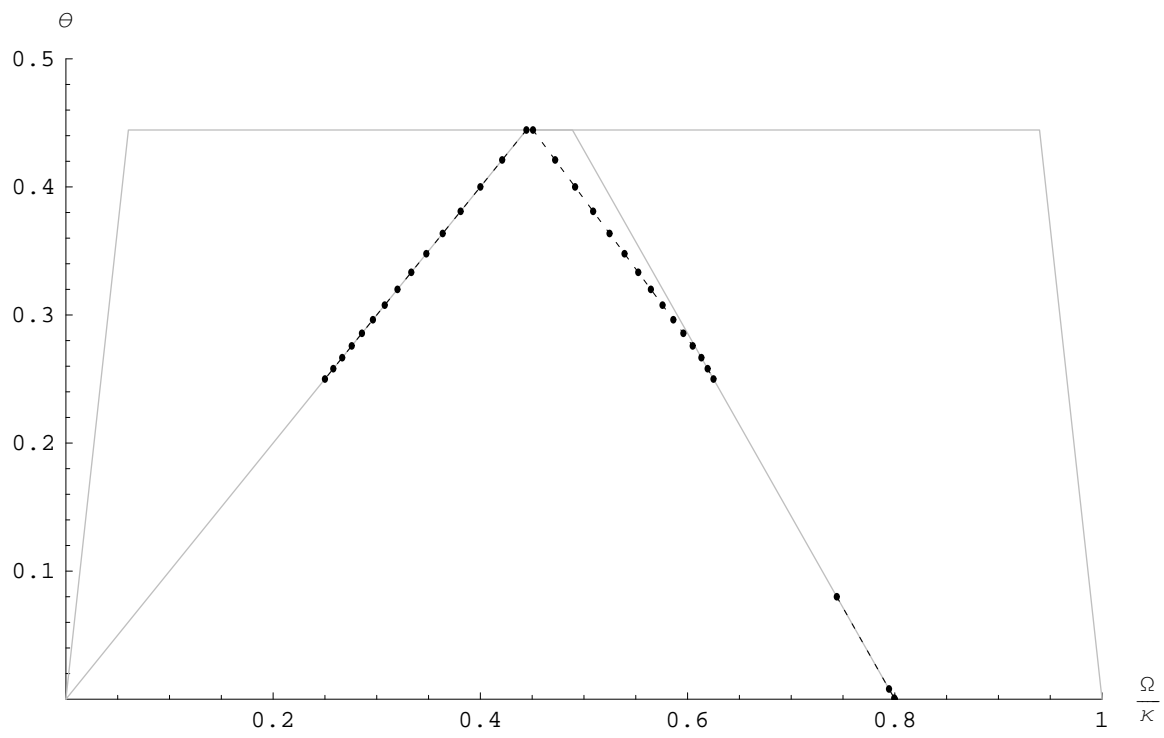


Figure 7.9: BBS_8

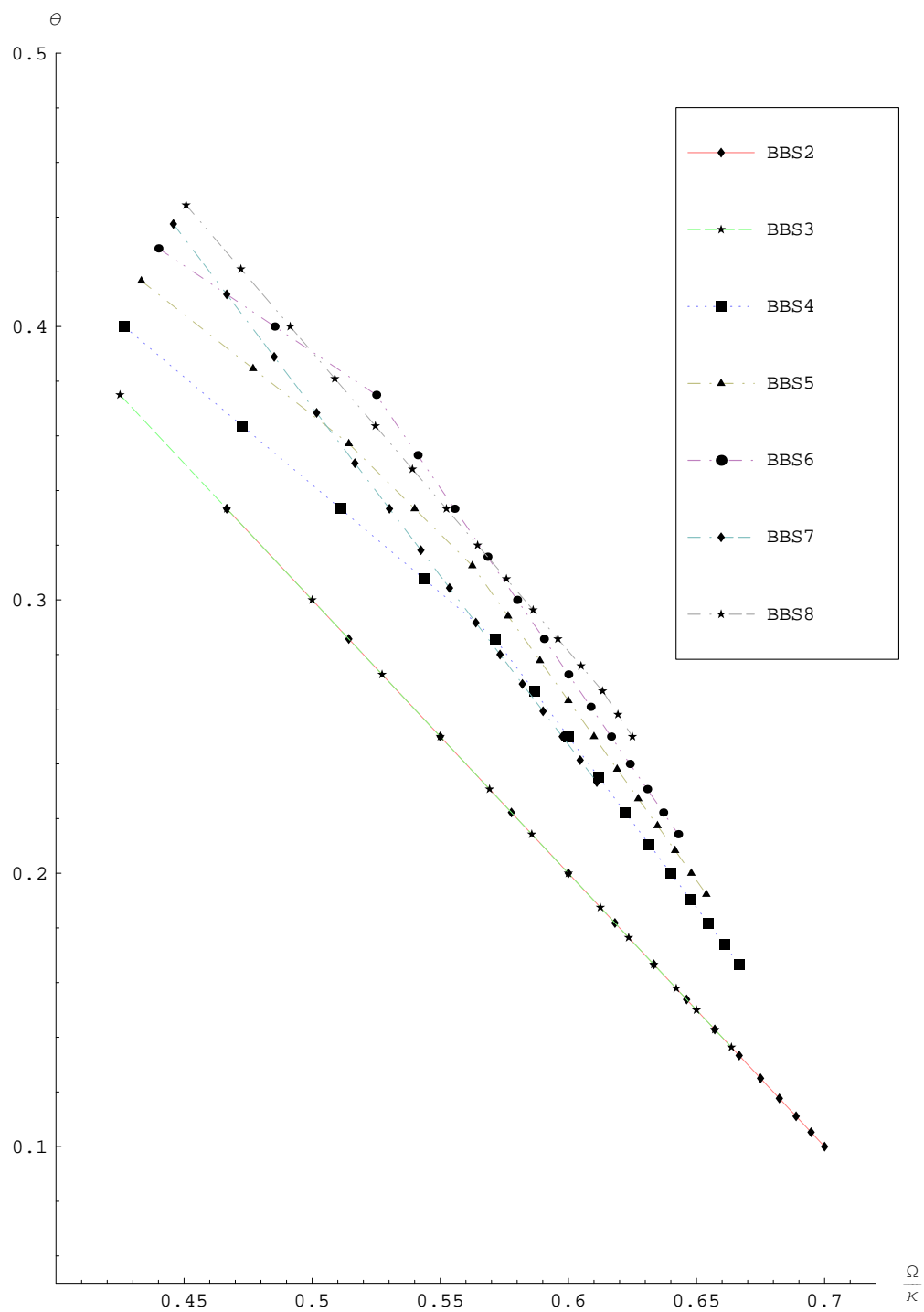


Figure 7.10: Comparison of the *BBS* systems

Chapter 8

Conclusion

The original intention of this thesis was to analyze a variety of block sorter designs. However, the outcome is that the first block sorter design, viz. the class of bubble block sorters, already provides such a rich behavior that only a thorough analysis of the class of bubble block sorter systems is given in this thesis.

We conclude this thesis with a detailed overview of the various steps that we have taken to reach our goal. Figure 8.1 gives a schematic overview of these steps and how they are related. The rectangular boxes represent the various stages in the process of the performance analysis and the solid arrows between these stages represent the methods or tools used to get from one stage to a next stage. The elliptical figures represent parameters and the dotted arrows show the connection between these parameters and the stage they are used in.

The main goal of this is to analyze performance, elasticity in particular, of various designs of stream processing systems that perform a block computation, e.g. the class of block sorter systems. These systems are defined by means of program texts. For this analysis, we use a method that depends on the computation of extremal schedules, mappings of the communication events of a system to time slots. We do not calculate the infinite schedules, but instead limit ourselves to the schedules of a finite number of consecutive events, starting with the first event on each port. Of course, this means that only finite behavior is described by these schedules. Naturally, we are more interested in the infinite behavior of these systems, however, we can use periodicity to extrapolate the results obtained from the finite behavior.

The extremal schedules we are interested in, can be computed by constructing and solving specific integer linear programming problems. Linear programming problems are defined by a finite number of decision variables, a linear objective (or cost) function that we want to minimize (or maximize) and linear constraints on the decision variables. These programming problems can be solved by an off the shelf linear programming solver and we used the mathematical application Mathematica for this purpose. We use the total latency for an initial number of blocks of a system that runs at fixed throughput as the objective function. The programming problems are parametrized with the throughput that we want to achieve. We want to both minimize and maximize this objective function. Furthermore, the constraints of the programming problem capture the ordering in which events can take place in the system. The decision variables of the programming problems are obtained from the precedence graph of a system. Each edge in the precedence graph can be associated with a single slack variable that is used as a decision variable in the programming problem. Furthermore, the constraints are also obtained from the precedence graph of a system.

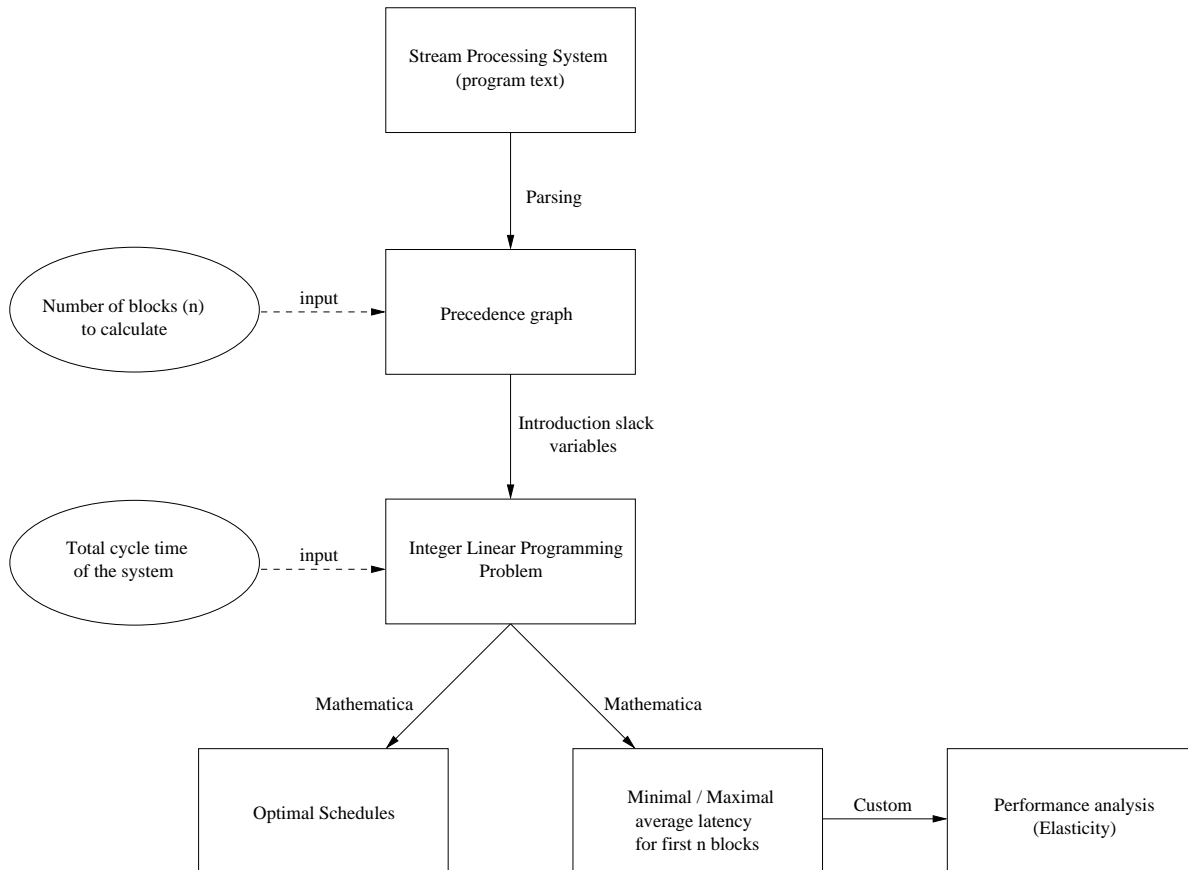


Figure 8.1: Schematic overview of the research.

The precedence graph of a system describes the ordering in which the events of a system can take place. Since the number of events of a stream processing system is infinite and we require a finite set of decision variables for our integer linear programming problem, we use a downward closed subgraph of the precedence graph that describes the precedence relation on the events of an initial number of blocks. The number of blocks that we want to calculate the precedence graph for, is a parameter.

We obtain the precedence graph of a particular design by parsing the program text of the system and constructing the precedence graph as an attribute of the resulting parse tree. A compiler, specially built for the research in this thesis, parses program text and constructs the associated integer linear programming problems. It depends on the attribute grammar found in Appendix A.

8.1 Future Research

We list several suggestions for further research and further additions to the tool.

- Further investigate bubble block sorter systems.
Although we already showed some very interesting results from the experiments we conducted, not all these results have been explained. Further research might shed light on why some bubble block sorter systems can achieve maximal average occupancy that is equal to the bound presented in [7], while other bubble block sorter systems can not achieve this bound.
- Compare various block sorter designs.
There are probably a lot of interesting properties that can be found by comparing various block sorter designs.
- Extend the tool to be able to handle a wider variety of basic components, for instance components with non-empty initialization.
If the tool can handle a larger set of components, other types of systems, e.g. FIR-filters, can be investigated.
- Extend the tool in such a manner that it can solve the integer programming problems on its own.
Since we use a general mathematical application to solve the integer linear programming problems, the time spent on solving these problems might be significantly more than the time spent by a solver implemented specifically for our problems.

Bibliography

- [1] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, Sept. 2000.
- [2] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [3] <http://www.python.org/doc/>. *Python Documentation Index*.
- [4] <http://www.wolfram.com/products/mathematica/index.html>. *Mathematica: The Way The World Calculates*.
- [5] J. Little. A proof of queueing formula: $L = \lambda W$. *Operations Research*, 9:383–387, 1961.
- [6] R. H. Mak. Periodic-drop-take calculus for stream transformers. Technical Report CS-Report 05-02, Technische Universiteit Eindhoven, May 2002.
- [7] R. H. Mak. Performance analysis of block computations with an application to block sorters, to be published. private note.
- [8] A. Peeters and M. de Wit. *Haste Manual*. 2004.
- [9] G. Sierksma. *Linear and integer programming: theory and practice*. Marcel Dekker, Inc., 1996.
- [10] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, July 1997.
- [11] T.A. Dolotta, S.B. Olsson and A.G. Petrucelli. *Unix User's Manual, Release 3.0*, June 1980.
- [12] H. ten Eikelder, R. van Geldrop, C. Hemerik, and G. Zwaan. *Syllabus bij het college COMPILERS (2M220)*. 1999.
- [13] G. Zwaan. *Parallel computations*. 1989.

Appendix A

Grammar

A.1 Attribute domains

| Attribute Name | Attribute type | Description |
|---------------------|--|---|
| <i>Name</i> | String | The set of all valid identifier names. |
| <i>Type</i> | String | The type of identifier, either 'in', 'out', 'chan', 'var' or 'val'. |
| <i>EType</i> | string | The type of an expression, either 'integer', 'boolean' or \square if it can be either. |
| <i>NameList</i> | List of <i>Name</i> | A list of names. |
| <i>NameTypeList</i> | List of (<i>Name</i> , <i>Type</i>) | A list containing the declared identifiers and their types. |
| <i>Syst</i> | $(Name \times \mathbb{N} \times \mathbb{N} \times \mathbb{N})$ | A tuple consisting of the name of a system and the number of input and output ports and value parameters. |
| <i>SystemList</i> | List of <i>Syst</i> | A list containing the declared systems and their input, output and value parameter count. |

A.2 Nonterminals (with attribute domains)

NAME $\langle +Name \rangle$
NUMBER $\langle +\mathbb{N} \rangle$
EXPRESSION $\langle +EType, -NameTypeList \rangle$
PREFIXOP $\langle +(EType, EType) \rangle$
E1 $\langle +EType, -NameTypeList \rangle$
REX1 $\langle -EType, +EType, -NameTypeList \rangle$
OP1 $\langle +(EType, EType) \rangle$
E2 $\langle +EType, -NameTypeList \rangle$
REX2 $\langle -EType, +EType, -NameTypeList \rangle$
OP2 $\langle +(EType, EType) \rangle$
E3 $\langle +EType, -NameTypeList \rangle$
REX3 $\langle -EType, +EType, -NameTypeList \rangle$

OP3 $\langle+(EType, EType)\rangle$
 E4 $\langle+EType, -NameTypeList\rangle$
 REX4 $\langle-EType, +EType, -NameTypeList\rangle$
 OP4 $\langle+(EType, EType)\rangle$
 E5 $\langle+EType, -NameTypeList\rangle$
 OP5 $\langle+(EType, EType)\rangle$
 E6 $\langle+EType, -NameTypeList\rangle$
 DENOT $\langle-NameTypeList\rangle$
 STATEMENT $\langle-NameTypeList\rangle$
 S1 $\langle-NameTypeList\rangle$
 S2 $\langle-NameTypeList\rangle$
 NAMELIST $\langle+NameList\rangle$
 INDECL $\langle+NameTypeList\rangle$
 OUTDECL $\langle+NameTypeList\rangle$
 CHANDECL $\langle+NameTypeList\rangle$
 VARDECL $\langle+NameTypeList\rangle$
 VALDECL $\langle+NameTypeList\rangle$
 COMPORSYSTDECL $\langle-SystemList, +Syst\rangle$
 COMPORSYST $\langle-SystemList, -Name, +Syst\rangle$
 COMP $\langle+NameTypeList\rangle$
 SYST $\langle-SystemList, -Name, +NameTypeList\rangle$
 PARAMDECL $\langle+NameTypeList\rangle$
 PORTDECL $\langle+NameTypeList\rangle$
 COMPBODY $\langle-NameTypeList\rangle$
 COMMAND $\langle-NameTypeList\rangle$
 SYSTBODY $\langle-SystemList, -NameTypeList\rangle$
 CASE $\langle-SystemList, -NameTypeList\rangle$
 GUARDEDBLOCK $\langle-SystemList, -NameTypeList\rangle$
 BLOCK $\langle-SystemList, -NameTypeList\rangle$
 COMPOSITION $\langle-SystemList, -NameTypeList\rangle$
 SYSTEMINSTANCE $\langle-SystemList, -NameTypeList\rangle$
 PARAMETERLIST $\langle-\mathbb{N}, -\mathbb{N}, -\mathbb{N}, -NameTypeList\rangle$
 PORTLIST $\langle-\mathbb{N}, -\mathbb{N}, -NameTypeList\rangle$
 VALLIST $\langle-\mathbb{N}, -NameTypeList\rangle$
 PARAMNAMELIST $\langle-\mathbb{N}, +NameList\rangle$
 START DECLARATION $\langle-SystemList, +SystemList\rangle$
 BODY $\langle-SystemList\rangle$

A.3 Terminals

The set of terminals is: { ' \leq ', '<', '=', ' \geq ', '>', '||', '|[' , '->', '-', ',, ';', '!', '?', '.', '(', ')', '>', '[', '[]', ']|', ']', '*', '&', '#', '#[', '+', 'case', 'comp', 'div', 'esac', 'max', 'min', 'mod', 'syst' }

A.4 Start symbol

The start symbol of the grammar is START.

A.5 Production rules and context conditions

We start with the production rules that give us names of identifiers and numbers.

NAME $\langle +n \rangle$::= IDENTIFIER $\langle +n \rangle$
 $n : n =$ the name of the identifier found.
 NUMBER $\langle +v \rangle$::= DIGIT { DIGIT }
 $v : v =$ the number represented by the concatenation of the digits.

The programming language has expressions. Expressions can be atomic, in case of an identifier or a number, a parenthesized expression or an operator applied to one or two operands, that are subexpressions. We do not have mixed type operators, meaning that the all operands are of the same type. Operators come in two flavors, infix operators (such as '+' and '\square if it can be either one (operator overloading). The resulting type is always 'integer' or 'boolean'. For instance, operator '\leq' expects integer operands and the result is a boolean. Note that the REX i nonterminals are introduced to make sure the grammar is LL(1)[12].

EXPRESSION $\langle +t, -ntl \rangle$::= E1 $\langle +t, -ntl \rangle$
 E1 $\langle +t, -ntl \rangle$::= E2 $\langle +t0, -ntl \rangle$
 REX1 $\langle -t0, +t, -ntl \rangle$::= OP1 $\langle +(t_{op}, t_{res}) \rangle$
 E1 $\langle +t1, -ntl \rangle$
 $: t0 = t1$
 $t : t = t_{res}$
 REX1 $\langle -t0, +t, -ntl \rangle$::= ϵ
 $t : t = t0$
 OP1 $\langle +(t_{op}, t_{res}) \rangle$::= '='
 $(t_{op}, t_{res}) : (t_{op}, t_{res}) = (\square, \text{'boolean'})$
 E2 $\langle +t, -ntl \rangle$::= E3 $\langle +t0, -ntl \rangle$
 REX2 $\langle -t0, +t, -ntl \rangle$::= OP2 $\langle +(t_{op}, t_{res}) \rangle$
 E2 $\langle +t1, -ntl \rangle$
 $: t0 = t_{op} \wedge t1 = t_{op}$
 $t : t = t_{res}$
 REX2 $\langle -t0, +t, -ntl \rangle$::= ϵ
 $t : t = t0$
 OP2 $\langle +(t_{op}, t_{res}) \rangle$::= '<'
 $(t_{op}, t_{res}) : (t_{op}, t_{res}) = (\text{'integer'}, \text{'boolean'})$
 OP2 $\langle +(t_{op}, t_{res}) \rangle$::= '>'

| | | |
|---|-----|---|
| OP2 $\langle+(t_{op}, t_{res})\rangle$ | ::= | $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'boolean')$ '<=' |
| OP2 $\langle+(t_{op}, t_{res})\rangle$ | ::= | $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'boolean')$ '>=' |
| E3 $\langle+t, -ntl\rangle$ | ::= | $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'boolean')$ E4 $\langle+t0, -ntl\rangle$ |
| REX3 $\langle-t0, +t, -ntl\rangle$ | ::= | REX3 $\langle-t0, +t, -ntl\rangle$ OP3 $\langle+(t_{op}, t_{res})\rangle$ E3 $\langle+t1, -ntl\rangle$: $t0 = t_{op} \wedge t1 = t_{op}$ $t : t = t_{res}$ |
| REX3 $\langle-t0, +t, -ntl\rangle$ | ::= | ϵ $t : t = t0$ |
| OP3 $\langle+(t_{op}, t_{res})\rangle$ | ::= | '+' $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |
| OP3 $\langle+(t_{op}, t_{res})\rangle$ | ::= | '-' $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |
| E4 $\langle+t, -ntl\rangle$ | ::= | E5 $\langle+t0, -ntl\rangle$ REX4 $\langle-t0, +t, -ntl\rangle$ |
| REX4 $\langle-t0, +t, -ntl\rangle$ | ::= | OP4 $\langle+(t_{op}, t_{res})\rangle$ E4 $\langle+t1, -ntl\rangle$: $t0 = t_{op} \wedge t1 = t_{op}$ $t : t = t_{res}$ |
| REX4 $\langle-t0, +t, -ntl\rangle$ | ::= | ϵ $t : t = t0$ |
| OP4 $\langle+(t_{op}, t_{res})\rangle$ | ::= | '*' $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |
| E5 $\langle+t, -ntl\rangle$ | ::= | OP5 $\langle+(t_{op}, t_{res})\rangle$ E5 $\langle+t0, -ntl\rangle$: $t0 = t_{op}$ $t : t = t_{res}$ |
| E5 $\langle+t, -ntl\rangle$ | ::= | PREFIXOP $\langle+(t_{op}, t_{res})\rangle$ '(' EXPRESSION $\langle+t0, -ntl\rangle$, , EXPRESSION $\langle+t1, -ntl\rangle$: $t0 = t_{op} \wedge t1 = t_{op}$ $t : t = t_{res}$ |
| E5 $\langle+t, -ntl\rangle$ | ::= | E6 $\langle+t, -ntl\rangle$ |
| PREFIXOP $\langle+(t_{op}, t_{res})\rangle$ | ::= | 'div' $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |
| PREFIXOP $\langle+(t_{op}, t_{res})\rangle$ | ::= | 'mod' $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |
| PREFIXOP $\langle+(t_{op}, t_{res})\rangle$ | ::= | 'min' $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |
| PREFIXOP $\langle+(t_{op}, t_{res})\rangle$ | ::= | 'max' $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |

| | | | |
|--|-----|----------------------------|--|
| OP5 $\langle+(t_{op}, t_{res})\rangle$ | ::= | '!' | $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('boolean', 'boolean')$ |
| OP5 $\langle+(t_{op}, t_{res})\rangle$ | ::= | '-' | $(t_{op}, t_{res}) : (t_{op}, t_{res}) = ('integer', 'integer')$ |
| E6 $\langle+t, -ntl\rangle$ | ::= | DENOT $\langle-ntl\rangle$ | $: t = 'integer'$ |
| E6 $\langle+t, -ntl\rangle$ | ::= | '(' | EXPRESSION $\langle+t, -ntl\rangle$ |
| | |)' | |
| DENOT $\langle-ntl\rangle$ | ::= | NAME $\langle+n\rangle$ | $: (\exists t :: (n, t) \in ntl)$ |
| DENOT $\langle-ntl\rangle$ | ::= | NUMBER $\langle+v\rangle$ | |

Apart from expressions, our language has statements. Communication actions, either read or write actions, are atomic statements. Furthermore, statements can be composed by sequentially or concurrently combining two sub statements. Concurrent composition of statements binds stronger than sequential composition, so $S1, S2; S3$ is evaluated as $(S1, S2); S3$. Apart from composing statements, we can group statements together by using parenthesis and we can repeat a (possibly composed) statement by using a repetition¹.

| | | | |
|--------------------------------|-----|---|--------------------------------------|
| STATEMENT $\langle-ntl\rangle$ | ::= | S1 $\langle-ntl\rangle$ | [|
| | | '; | STATEMENT $\langle-ntl\rangle$ |
| | |] | |
| S1 $\langle-ntl\rangle$ | ::= | S2 $\langle-ntl\rangle$ | [|
| | | '; | S1 $\langle-ntl\rangle$ |
| | |] | |
| S2 $\langle-ntl\rangle$ | ::= | NAME $\langle+n0\rangle$ '?' | NAME $\langle+n1\rangle$ |
| | | $: (n0, 'in') \in ntl$ | |
| | | $: (n1, 'var') \in ntl$ | |
| S2 $\langle-ntl\rangle$ | ::= | NAME $\langle+n0\rangle$ '!' | EXPRESSION $\langle+t, -ntl\rangle$ |
| | | $: (n0, 'out') \in ntl$ | |
| | | $: t = 'integer'$ | |
| S2 $\langle-ntl\rangle$ | ::= | '(' | STATEMENT $\langle-ntl\rangle$ |
| | |)' | |
| S2 $\langle-ntl\rangle$ | ::= | '#' | EXPRESSION $\langle+t, -ntl0\rangle$ |
| | | '[' | STATEMENT $\langle-ntl\rangle$ |
| | | ']' | |
| | | $ntl0 : ntl0 = \{(n, 'val') (n, 'val') \in ntl\}$ | |

¹This repetition is merely an abbreviation and at 'compile time' repetitions are substituted with the actual statements.

$: t = \text{'integer'}$

The language has constructions to declare variables of a certain type. We have identifiers denoting input and output ports ('in' and 'out' respectively), channels ('chan'), variables that can store read values ('var') and value parameters ('val') that are used to define whole classes of components or systems at once. A declaration starts with a keyword (for instance 'chan'), followed by a list of distinct names, separated by commas.

$$\begin{aligned}
 \text{NAMELIST}\langle +ntl \rangle & ::= \text{NAME}\langle +n \rangle \\
 & \quad [\\
 & \quad \quad \text{' , '}, \\
 & \quad \quad \text{NAMELIST}\langle +nl0 \rangle \\
 & \quad] \\
 & \quad : n \notin nl0 \\
 & \quad nl : nl = [n] \oplus nl0 \\
 \text{INDECL}\langle +ntl \rangle & ::= \text{'in'} \\
 & \quad \text{NAMELIST}\langle +ipl \rangle \\
 & \quad : (\forall n0, n1 : n0, n1 \in ipl : n0 \neq n1) \\
 & \quad ntl : ntl = \{(n, in) | n \in ipl\} \\
 \text{OUTDECL}\langle +ntl \rangle & ::= \text{'out'} \\
 & \quad \text{NAMELIST}\langle +opl \rangle \\
 & \quad : (\forall n0, n1 : n0, n1 \in opl : n0 \neq n1) \\
 & \quad ntl : ntl = \{(n, out) | n \in opl\} \\
 \text{CHANDECL}\langle +ntl \rangle & ::= \text{'chan'} \\
 & \quad \text{NAMELIST}\langle +chanl \rangle \\
 & \quad : (\forall n0, n1 : n0, n1 \in chanl : n0 \neq n1) \\
 & \quad ntl : ntl = \{(n, chan) | n \in chanl\} \\
 \text{VARDECL}\langle +ntl \rangle & ::= \text{'var'} \\
 & \quad \text{NAMELIST}\langle +varl \rangle \\
 & \quad : (\forall n0, n1 : n0, n1 \in varl : n0 \neq n1) \\
 & \quad ntl : ntl = \{(n, var) | n \in varl\} \\
 \text{VALDECL}\langle +ntl \rangle & ::= \text{'val'} \\
 & \quad \text{NAMELIST}\langle +vall \rangle \\
 & \quad : (\forall n0, n1 : n0, n1 \in vall : n0 \neq n1) \\
 & \quad ntl : ntl = \{(n, val) | n \in vall\}
 \end{aligned}$$

In the programming language we can declare components and systems. These declarations are separated into the heading and the body of the declaration. The heading starts with the name of the component or system, followed by keyword 'comp' or 'syst' for respectively a component or system declaration, followed by the parameter declaration. The parameter declaration declares both the input and output ports and (optionally) the value parameters. The body part of system and component declarations are discussed below.

$$\begin{aligned}
 \text{COMPORSYSTDECL}\langle -systs, +syst \rangle & ::= \text{NAME}\langle +n \rangle \\
 & \quad \text{' = '}, \\
 & \quad \text{COMPORSYST}\langle -systs, -n, +syst \rangle \\
 & \quad : (\forall n0, i, o, v : (n0, i, o, v) \in systs : n \neq n0)
 \end{aligned}$$

| | | |
|--|-----|---|
| COMPORSYST $\langle -systs, -n, +syst \rangle$ | ::= | COMP $\langle +ntl \rangle$ with $f(val, list) = (\#i :: (i, val) \in list)$ $syst : syst = (n, f('in', ntl), f('out', ntl), f('val', ntl))$ |
| COMPORSYST $\langle -systs, -n, +syst \rangle$ | ::= | SYST $\langle -systs, -n, +ntl \rangle$ with $f(val, list) = (\#i :: (i, val) \in list)$ $syst : syst = (n, f('in', ntl), f('out', ntl), f('val', ntl))$ |
| COMP $\langle +ntl \rangle$ | ::= | 'comp' PARAMDECL $\langle +ntl \rangle$ COMPBODY $\langle -ntl \rangle$ |
| SYST $\langle -systs, -n, +ntl \rangle$ | ::= | 'syst' PARAMDECL $\langle +ntl \rangle$ SYSTBODY $\langle -systs0, -ntl \rangle$ with $f(val, list) = (\#i :: (i, val) \in list)$ $syst = (n, f('in', ntl), f('out', ntl), f('val', ntl))$ $systs0 : systs0 = systs \oplus [syst]$ |
| PARAMDECL $\langle +ntl \rangle$ | ::= | ' (' PORTDECL $\langle +ntl0 \rangle$ ['&' VALDECL $\langle +ntl1 \rangle$])' '.' $ntl : ntl = ntl0 \oplus ntl1$ |
| PORTDECL $\langle +ntl \rangle$ | ::= | INDECL $\langle +ipl \rangle$ '&' OUTDECL $\langle +opl \rangle$: $(\forall n0, n1 : (n0, in) \in ipl \wedge (n1, out) \in opl : n0 \neq n1)$ $ntl : ntl = ipl \oplus opl$ |
| PORTDECL $\langle +ntl \rangle$ | ::= | INDECL $\langle +ipl \rangle$ $ntl : ntl = ipl$ |
| PORTDECL $\langle +ntl \rangle$ | ::= | OUTDECL $\langle +opl \rangle$ $ntl : ntl = opl$ |

The body of a component declaration starts with a declaration of the variables, followed by a command, which consists of a (possibly empty) initialization statement followed by an infinite repetition of a statement.

| | | |
|----------------------------------|-----|---|
| COMPBODY $\langle -ntl0 \rangle$ | ::= | ' [' VARDECL $\langle +ntl1 \rangle$ '>' COMMAND $\langle -ntl \rangle$ ']' : $(\forall n0, t0, n1, t1 : (n0, t0) \in ntl0 \wedge (n1, t1) \in ntl1 : n0 \neq n1)$ $ntl : ntl = ntl0 \oplus ntl1$ |
| COMMAND $\langle -ntl \rangle$ | ::= | [|

```

    STATEMENT⟨-ntl⟩
  ]
  '#['
  STATEMENT⟨-ntl⟩
  ']'

```

The body of a system declaration can consist of either one block or a case distinction on (one of) the value parameters. Each case consists of a block. And a block (optionally) starts with a channel declaration, followed by the composition of one or more instances of (previously declared) systems. An instance of a system starts with the name of the system, followed by the actual parameters of the system.

```

SYSTBODY⟨-syssts, -ntl⟩ ::= BLOCK⟨-syssts, -ntl⟩
SYSTBODY⟨-syssts, -ntl⟩ ::= CASE⟨-syssts, -ntl⟩
CASE⟨-syssts, -ntl⟩ ::= 'case'
                       GUARDEDDBLOCK⟨-syssts, -ntl⟩
                       {
                         '['
                         GUARDEDDBLOCK⟨-syssts, -ntl⟩
                       }
                       'esac'
GUARDEDDBLOCK⟨-syssts, -ntl⟩ ::= EXPRESSION⟨+t, -ntl0⟩
                              '->'
                              BLOCK⟨-syssts, -ntl⟩
                              : t = 'boolean'
                              ntl0 : ntl0 = {(n, val) | (n, val) ∈ ntl}
BLOCK⟨-syssts, -ntl0⟩ ::= '['
                       [
                         CHANDECL⟨+ntl1⟩
                         '>'
                       ]
                       COMPOSITION⟨-syssts, -ntl⟩
                       ']'
                       : (∀n0, t0, n1, t1 : (n0, t0) ∈ ntl0 ∧ (n1, t1) ∈ ntl1 : n0 ≠ n1)
                       ntl : ntl = ntl0 ⊕ ntl1
COMPOSITION⟨-syssts, -ntl⟩ ::= SYSTEMINSTANCE⟨-syssts, -ntl⟩
                              [
                                '||'
                                COMPOSITION⟨-syssts, -ntl⟩
                              ]
SYSTEMINSTANCE⟨-syssts, -ntl⟩ ::= NAME⟨+n⟩
                              '('
                              PARAMETERLIST⟨-i, -o, -v, -ntl⟩
                              ')'
                              i, o, v : (n, i, o, v) ∈ syssts
PARAMETERLIST⟨-i, -o, -v, -ntl⟩ ::= PORTLIST⟨-i, -o, -ntl⟩
                                  : v = 0

```

| | | |
|--|-----|--|
| PARAMETERLIST $\langle -i, -o, -v, -ntl \rangle$ | ::= | PORTLIST $\langle -i, -o, -ntl \rangle$ VALLIST $\langle -v, -ntl \rangle$: $v > 0$ |
| PORTLIST $\langle -i, -o, -ntl \rangle$ | ::= | PARAMNAMELIST $\langle -i, +nl \rangle$: $i > 0 \wedge o = 0$: $(\forall n : n \in nl : (n, 'in') \in ntl \vee (n, 'chan') \in ntl)$ |
| PORTLIST $\langle -i, -o, -ntl \rangle$ | ::= | PARAMNAMELIST $\langle -o, +nl \rangle$: $i = 0 \wedge o > 0$: $(\forall n : n \in nl : (n, 'out') \in ntl \vee (n, 'chan') \in ntl)$ |
| PORTLIST $\langle -i, -o, -ntl \rangle$ | ::= | PARAMNAMELIST $\langle -i, +nl0 \rangle$, , PARAMNAMELIST $\langle -o, +nl1 \rangle$: $i > 0 \wedge o > 0$: $(\forall n : n \in nl0 : (n, 'in') \in ntl \vee (n, 'chan') \in ntl)$: $(\forall n : n \in nl1 : (n, 'out') \in ntl \vee (n, 'chan') \in ntl)$ |
| VALLIST $\langle -v, -ntl \rangle$ | ::= | EXPRESSION $\langle +t, -ntl \rangle$, , VALLIST $\langle -(v-1), -ntl \rangle$: $v > 1$ |
| VALLIST $\langle -v, -ntl \rangle$ | ::= | EXPRESSION $\langle +t, -ntl \rangle$: $v = 1$ |
| PARAMNAMELIST $\langle -num, +nl \rangle$ | ::= | NAME $\langle +n \rangle$, , PARAMNAMELIST $\langle -(num-1), +nl0 \rangle$: $num > 1$: $n \notin nl0$ $nl : nl = [n] \oplus nl0$ |
| PARAMNAMELIST $\langle -num, +nl \rangle$ | ::= | NAME $\langle +n \rangle$: $num = 1$ $nl : nl = [n]$ |

Finally we have a declaration part in which components and systems are declared. It is followed by a block containing a list of input ports, a list of output ports and one instance of a system.

| | | |
|--|-----|---|
| START | ::= | DECLARATION $\langle -sys0, +sys1 \rangle$ BODY $\langle -sys1 \rangle$ $sys0 : sys0 = \emptyset$ |
| DECLARATION $\langle -sys0, +sys1 \rangle$ | ::= | COMPORSYSTDECL $\langle -sys0, +sys \rangle$ DECLARATION $\langle -sys2, +sys1 \rangle$ $sys2 : sys2 = sys0 \cup \{sys\}$ |
| DECLARATION $\langle -sys0, +sys1 \rangle$ | ::= | ϵ $sys1 : sys1 = sys0$ |
| BODY $\langle -sys \rangle$ | ::= | ' [INDECL $\langle +intl \rangle$ ' & ' OUTDECL $\langle +ontl \rangle$ |

'[>'
SYSTEMINSTANCE(-sys t s, -ntl)
'|'
: ($\forall n_0, n_1 : (n_0, \text{'in'}) \in \text{intl} \wedge (n_1, \text{'out'}) \in \text{ontl} : n_0 \neq n_1$)
ntl : ntl = intl \oplus ontl

Appendix B

Tokens

| Token | Description |
|-------------------------------------|---|
| EOFToken | The end of file token |
| IDToken | Token representing an identifier (a letter followed by zero or more letters, numbers or underscores, but not a reserved word) |
| NumToken | Token representing a positive integer |
| MinToken | Token for reserved word 'min' |
| MaxToken | Token for reserved word 'max' |
| DivToken | Token for reserved word 'div' |
| ModToken | Token for reserved word 'mod' |
| VarToken | Token for reserved word 'var' |
| ChanToken | Token for reserved word 'chan' |
| ValToken | Token for reserved word 'val' |
| InToken | Token for reserved word 'in' |
| OutToken | Token for reserved word 'out' |
| SystToken | Token for reserved word 'syst' |
| CompToken | Token for reserved word 'comp' |
| CaseToken | Token for reserved word 'case' |
| EsacToken | Token for reserved word 'esac' |
| PlusSym | A token for plus symbol '+' |
| MinSym | A token for minus symbol '-' |
| LessThanSym | A token for the less than symbol '<' |
| GreaterThanSym | A token for the greater than symbol '>' |
| TimesSym | A token for the times symbol '*' |
| IsSym | A token for equal symbol '=' |
| BlockStartSym | A token for the block start symbol ' [' |
| BlockEndSym | A token for the block end symbol '] ' |
| InfiniteRepetitionStartToken | A token for the infinite repetition start symbol '#[' |
| PoundSym | A token for the pound symbol (not followed by '[') |
| OpenParSym | A token for the left parenthesis symbol '(' |
| CloseParSym | A token for the right parenthesis symbol ')' |

| | |
|-------------------------|--|
| OpenBrackSym | A token for the left bracket symbol '[' (not followed by '>' or ']') |
| CloseBrackSym | A token for the right bracket symbol ']' |
| CommaSym | A token for the comma symbol ',' |
| DotSym | A token for the period symbol '.' |
| SemicolonSym | A token for the semicolon symbol ';' |
| QmarkSym | A token for the question mark symbol '?' |
| EmarkSym | A token for the exclamation mark symbol '!' |
| AmpSym | Token representing the ampersand '&' |
| SeperatorSym | A token for the seperation symbol '>' |
| CaseSeperatorSym | A token for the seperation symbol '[]' |
| ArrowSym | A token for the seperation symbol '->' |
| ParallelSym | A token for the parallel composition symbol ' ' |