High Level Verification
of
(A)synchronous Circuit Descriptions

G.G. de Jong

Master thesis
reporting on graduation work
performed from 2.04.87 to 16.12.87
by order of prof. dr.-ing. J.A.G. Jess
and supervised by ir. L. Stok
Abstract

In the development of a silicon compiler, one has to choose if synchronous or self-timed systems should be generated. In the silicon compiler EASY, which is in development in this department, synchronous systems will be synthesised automatically. However, for large systems it may be profitable to split up the whole system into several smaller systems which interact with eachother. This interaction is in fact a matter of communication and this communication happens in an asynchronous way. Protocols have to be designed for this interaction to assure the correct behaviour of the whole system. Therefore, such a protocol and the environment it is working in must be verified.

Petri nets have been proven very useful in the verification of communication protocols. In that case, the verification can be done with help of petri net analysis methods. This method suits also very well for the silicon compiler, since the data structure of the higher levels of the silicon compiler can be mapped one to one onto a petri net. Also, this data structure needs only a slight extension to model asynchronous features. Therefore, as well as the functional description of the system as the protocols can be written in the same way and this results in just one model which has to be verified.

This method of verification gives good results. However, it is a rather space and time consuming analysis method, since all possible states of the system have to be computed. Therefore, some efforts are made for reducing the complexity of the state space. Since only that states which give new information about the behaviour of the system need to be computed, it is useful not to compute the redundant states. Such reductions are proposed and implemented. The result is a far less space and time consuming method for analysing petri nets. Therefore, this method can be used for the verification of the behaviour of a set of interacting systems in the silicon compiler.

The verification does not only give an answer to the question whether a specified protocol has a correct behaviour or not, but may also give some hints for constraints. When these constraints are applied to the system, the behaviour of the protocol and its environment may be correct. These constraints have to be passed to the hardware synthesis phase of the silicon compiler to generate a circuit which fulfills the functional description.
CONTENTS

1. Introduction 1

2. Design methods for integrated circuits 3
   2.1 Synchronous circuits 3
   2.2 Self-timed circuits 4
   2.3 Automatic generation of circuits 5
   2.4 Splitting up the circuit in parts 5
   2.5 Summary 6

3. Demand graphs and petri nets 8
   3.1 Asynchronous elements of the demand graph 9
   3.2 Petri nets 11
   3.3 Mapping of demand graphs onto petri nets 12

4. Petri nets 15
   4.1 Mathematical definition of a petri net 15
   4.2 Graphical representation of a petri net 15
   4.3 Definitions for the behaviour of a petri net 15
   4.4 Modelling with petri nets 18
   4.5 Analysis of petri nets 20
   4.6 Reductions 26
   4.7 Petri nets with time 31

5. Implementation of the verification of systems 39
   5.1 Conversion of demand graph into a petri net 39
   5.2 Analysis of the petri net 40
   5.3 Implementation of markings 47
   5.4 Complexity of the construction of the reachability tree 47

6. Results 49
   6.1 Tseng example 49
   6.2 Simple communication example 49
   6.3 Two phase handshaking protocol 50

7. Conclusions 52

References 55

Appendix 1: Tseng Example 57
Appendix 2: Simple communication example without any protocol 63
Appendix 3: A two phase handshaking protocol 68
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overview of a silicon compiler</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>a) a while loop b) a on statement or a waiting loop</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>The demand graph of the example</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Interacting systems a) demand graph representation b) petri net representation</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>A petri net a) descriptive b) graphical</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Firing of a petri net a) before firing transition $t_1$ b) after firing transition $t_1$</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>Concurrency, all transitions may fire in any order.</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>A conflict situation, firing one transition disables the other.</td>
<td>19</td>
</tr>
<tr>
<td>4.5</td>
<td>A transition which may not fire if tokens in the output places exist</td>
<td>20</td>
</tr>
<tr>
<td>4.6</td>
<td>An infinite reachability tree for a petri net with a finite reachability set.</td>
<td>23</td>
</tr>
<tr>
<td>4.7</td>
<td>a) A petri net b) The reachability tree</td>
<td>27</td>
</tr>
<tr>
<td>4.8</td>
<td>a) a petri net b) the reachability tree</td>
<td>28</td>
</tr>
<tr>
<td>4.9</td>
<td>Well-behaved petri net with well-behaved subnet</td>
<td>30</td>
</tr>
<tr>
<td>4.10</td>
<td>A petri net, which is not well-behaved, with well-behaved subnet</td>
<td>31</td>
</tr>
<tr>
<td>4.11</td>
<td>Transformation of a place of a timed place transition net into a subnet of a timed petri net.</td>
<td>32</td>
</tr>
<tr>
<td>4.12</td>
<td>Transformation of a transition of a timed petri net into a subnet of a timed place transition net.</td>
<td>33</td>
</tr>
<tr>
<td>4.13</td>
<td>Transformation of transition in a timed petri net into a subnet of a timed petri net.</td>
<td>33</td>
</tr>
<tr>
<td>4.14</td>
<td>Example of timing constraints caused by the firing of transition $t_1$</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Reduction of an operator network a) original network b) reduced network</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Removal of link nodes in the petri net a) demand graph b) corresponding petri net</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>Two communication cycles in a two phase handshaking protocol</td>
<td>50</td>
</tr>
<tr>
<td>6.2</td>
<td>One communication cycle in a four phase handshaking protocol</td>
<td>51</td>
</tr>
</tbody>
</table>
1. Introduction

Nowadays, many systems are designed as integrated circuits. These integrated circuits perform the same operation as other circuits, for instance printed circuit boards, but are much smaller. Since the technology to make integrated circuits is capable to design large circuits on a small silicon area, the systems may be very complex. This results in Very Large Scale Integrated Circuits. Since designing such complex systems involves a very large amount of time, one should design such systems, or parts of these systems, automatically.

The automatical synthesis of integrated circuits may be done on several levels. For instance, the layout of a system may be automatically generated from the logical description of the system. Such a logical description of a system consists of the logic gates which are necessary and the interconnection of these gates. In this case, the designer must take care of specifying a logical description. Since most designs start with a description of the (global) behaviour of the system, this description must be converted to the logical description. It would be nice if this translation can also be performed automatically.

The task of a silicon compiler is to convert a functional description of a system to a layout of a circuit. The functional description describes the behaviour of the system. At the automatic system design group the silicon compiler EASY, Eindhoven Architecture Synthesis, is in development. Since the automatically generated circuit must behave according to its functional description, verification of the behaviour is necessary. Two ways to assure the correct behaviour of the generated circuits exist. The first method is to extract the behaviour of the circuit and compare it with the specified behaviour. If these two behaviours are not identical, then a circuit has been generated which is not meant by the designer. The other approach for verification is no verification at all but generation of a circuit which behaves 'correct by construction'. This means, that all steps involved in the transformation of a functional description to the layout of the circuit result in an intermediate data structure which has the same behaviour as the functional description.

Of course, the second method of verification is more favourable, since the resulting circuit is in general quite large. Therefore, verification of the circuit will be quite expensive and time consuming. Another possibility could be to verify after each step of the silicon compilation. In that case, the end result of a step is verified against the input specification of this step. In this case, several verifications have to be made but for which the sum is hopefully less than the one verification at the lowest level.

In most systems, one can distinguish several parts which have their own function and which are independent of the other parts. The composition of all these subsystems constitutes the whole system. Eventually, these subsystems can also be divided into sub-subsystems in the same way. This results in a hierarchical modular system. Of course, the subsystems are not totally independent of each other. Otherwise, they would not be necessary since they have no effect on the total behaviour of the system. Therefore, these subsystems communicate with each other. The behaviour of the whole systems depends on the behaviour of the separate subsystems and on the interaction between them.

When the system is divided in parts, one has to make clear how these parts interact with each other. The interaction is established by communication of the subsystems. For example, one subsystem asks data from another subsystem. This other subsystem must provide this data and tell the first system that the data is available. Also it has to be made clear in what places this data is available. All these things are represented in a protocol. Such a protocol defines the behaviour of the interaction between two or more systems. Of course, these protocols must fulfill certain conditions. Such a condition may be that no messages are clobbered. This means that no new message may be sent if
the previous message has not arrived. Therefore, verification of the correct behaviour of protocols is needed.

The verification of such protocols is the main subject of this report. This verification is meant to be built in the EASY silicon compiler. The verification gives feedback to the user, for instance suggestions for timing constraints. The user may implement these constraints in the functional description in such a way that the silicon compiler can synthesise a correct hardware realisation.
2. Design methods for integrated circuits

Two major methods for designing integrated circuits exist. One may generate *synchronous* circuits or *asynchronous* circuits. Asynchronous circuits are also called *self-timed* circuits. In synchronous circuits, a clock indicates to the separate elements to perform the action of that element. Therefore, only at clock ticks actions are initiated. Of course, the action must have finished before the next clock tick. So, an absolute time measure is introduced by the clock of the circuit. In asynchronous or self-timed circuits, no absolute time measure is available, since each element takes care of the time it must perform the action. Also, the element itself indicates the end of the action. So, a time measure is only defined by the elements and only an ordering of action can be given, but no information about the real time. Time can only be made explicit in self-timed circuits when the exact propagation times of the separate elements are investigated.

2.1 Synchronous circuits

In synchronous systems circuits, actions may only be initiated at the times indicated by the clock. Thus, one does not have to consider extensively the timing relations between the separate elements. Therefore, most circuits are designed as synchronous circuits. The signal transitions, caused by the action of an element, are between two clock ticks. These transitions are propagated to other elements at the next clock tick. Of course, the element should reach a correct and stable state at the next clock tick. In principle, this timing constraint is the only constraint for the timing behaviour of an element. Therefore, the clock rate may never be larger than the maximal delay of all elements for a correct behaviour of the circuit. A smaller clock rate is possible, since it will be sure that at each clock tick the state of the circuit is stable and correct according to the function of the element. A faster clock is not allowed, since it will not always be sure that the data is correct or stable at the next clock tick.

However, other timing problems may be encountered in designing synchronous circuits. The clock must be well defined. The clock must not only have the proper period time, but also the active clock period must be well defined. The active clock period is the time the clock is high, so that it can enable the elements according to the preset and hold times of the elements. These preset and hold times need also to be considered when the maximal clock rate is computed.

Clock skew is another problem in synchronous systems. Especially in large circuits, the wires of the clock signal become long and introduce some delay in the clock signal. This delay is not only due to the fact that the signal travels with a certain speed along the wire but also due to the resistance and capacitance of the wire. Therefore, a slight difference exists in the same clock tick in different parts of the circuit. This difference of clock pulses in different parts of the circuit, the so-called clock skew, may cause incorrect behaviour since the timing constraints can be disobeyed.

This clock skew may be solved by dividing the whole circuit in parts. Each part or subcircuit has its own clock, which is independent of the clocks in the other subcircuits. The subcircuits are designed in such a way that no clock skew occurs or that it does not have influence on the behaviour of the subcircuit. This division of the whole circuit into parts is also done for designing the circuit itself, since most circuits are designed in a modular way. Therefore, these two divisions may be done at the same time and may be identical.

However, communication between subcircuits exist, otherwise a subcircuit has no useful meaning in the whole circuit. The incoming data has to be synchronised with the local clock of the subcircuit. A synchroniser can be seen as an element with two inputs: one for the clock of the system and one for the data which must be synchronised. The synchronisation means that the data must be input to the system at the clock.
ticks and not on any other time. However, problems may arise when the clock and the data arrives at the same time. In that case the output is unpredictable, since the synchroniser may get into a metastable state. Therefore, one can never assure 100% reliability in synchronising two signals. The time in which an element stays in a metastable state is not known. This time may even be unbounded. Therefore, it is not known whenever the synchroniser results in the correct state and it is certainly not related to the clock of the subcircuit. Such a metastable state of the synchroniser may result in a behaviour which should never happen.

This problem may also arise when the communication of the whole circuit with the outside world, i.e. the environment of the circuit, is concerned. In that case, a synchroniser is needed for the data coming from the environment, since it is not exactly known at which time data is given to the circuit.

With more complex circuits, one may reduce the probability that the synchroniser gets into a metastable state. These circuits will cost time and chip area and therefore reduce the efficiency of the circuit. However, the metastable state of a synchroniser can probably never be eliminated by a continuous running clocked. However, when the clock of the subcircuit can be started asynchronously, one can eliminate the metastable state of the synchroniser. \cite{Seit80} and \cite{Pech76}

2.2 Self-timed circuits

Asynchronous or self-timed circuits have no problems with synchronisation of two data streams, since the elements themselves take care when to perform their action. These actions are not defined to occur at a certain time, but depend on the actions of the environment. Therefore, only the order of events is known but not the exact time of the occurrence of an event. An action of an element will be initiated whenever its inputs tell the element to do so. If the action is finished, the element gives its environment a ready signal. This ready signal may be used by another element to initiate its action.

Since self-timed circuits react on changes of signal values. Therefore, these self-timed circuits are very sensitive for spikes. Spikes are signal value changes which durate only a short time and may occur especially when signals are not stable. Thus, self-timed elements must be carefully designed to avoid spikes. This is a very delicate problem and each element has to be fully tested for correct behaviour. In synchronous circuits, such strict constraints are not necessary, since no new action will be performed before the next clock tick.

A great disadvantage of asynchronous systems is the fact that each element has a protocol mechanism, for instance two or four phase handshaking, to assure the self-timed nature of the element, since in general an element is not self-timed by itself. Only elements like C-Muller elements\footnote{A C-Muller element is a kind of 'rendez-vous'. The output of the C-Muller element only changes, whenever both inputs have changed to the same value.} are self-timed already. Therefore, each element is larger than necessary. Also, such a protocol needs time. Also, each connection has in principle to be made double or even triple: one connection for the data, two for the protocol (one line to the other element and one coming from that element). This takes a large amount of space in the circuit. However, when data busses are concerned, not all data lines need to be doubled if an appropriate coding is applied to the data. Thus, many extensions are needed for an element which all cost space and time. First experiments with designing self-timed circuits result in a 1.5 to 2 times larger circuit than the synchronous equivalent. \cite{Rem87}
Also, self-timed elements may be made as synchronous elements. However, these elements must be imbedded in a circuit which fulfills the self-timed constraints [List85a], [List85b]. This means that a protocol is specified for each element and this results also in the previous described disadvantages.

2.3 Automatic generation of circuits

When a circuit has to be designed automatically from a functional description and the circuit has to fulfill some constraints, it is less difficult to do this for synchronous systems than for self-timed systems. In most cases, a synchronous circuit will be designed without precise knowledge about the timing, since everything happens on clock ticks. Therefore, a controller takes care of activating a module on the right time for the proper action with the appropriate data. This controller does its job for each clock period. In self-timed circuits, it is more complicated, since each element takes care of performing its own action at a particular time. Scheduling of tasks is also much more difficult for self-timed circuits than for synchronous circuits, since time slots at which something takes place are not known. Because of the self-timed nature of asynchronous systems, a controller is not needed in these systems.

In synchronous systems, specified timing constraints can be obeyed by the controller. In self-timed circuits this is much more difficult since the exact timing is not known.

Other constraints, like chip area, can also be dealt with more easily by synchronous systems than by self-timed systems. These constraints may also be handled by the controller or scheduler. For example, if the chip area should be as small as possible, the scheduler may map all identical operations onto the same module, since the scheduler knows at which time a module is idle. This is not the case for self-timed systems, since there is no explicit scheduler.

Self-timed circuits can be designed automatically. In general, the functional description of the circuit is mapped one to one onto hardware modules. The whole circuit consists of many basic hardware elements. These basic elements are, of course, self-timed. A composition of these elements assure that the whole system is self-timed. If the functional description follows some (syntactical) rules, it can be proved that the resulting physical circuit behaves like the functional description [Eber87].

It is clear that no constraints, like timing or area constraints, can be dealt with in such self-timed circuits, except if these constraints are implicit in the specified functional behaviour. One could tell something about the timing of the circuit only after the design is made. The same applies to the area of the circuit. If the result is not satisfactory, the functional description must be changed in such a way that the wished constraints are valid. For example, if the circuit consumes too much chip area, the functional description has to be made shorter, since the space requirements for a circuit are proportional to the length of the functional description. So, the user has to find another functional description which has the same behaviour.

It is clear, that constraints have to be implemented implicitly in the functional description by the user when asynchronous systems are designed. However, synthesis tools for synchronous systems can deal with explicit constraints, i.e. constraints which are specified independently of the structure of the functional description.

2.4 Splitting up the circuit in parts

As mentioned before, it is reasonable to split up a VLSI circuit into subcircuits. Each subcircuit performs a part of the functional description and communicates with the other subcircuits. The subcircuits should be designed as synchronous circuits and not as self-timed circuits for reasons of simplicity. Each synchronous subcircuit has its own clock independent of the clocks in the other subcircuits. Therefore, the
communication between subcircuits can be best done asynchronously.

The communication of subsystems must behave correctly. Therefore, a protocol has to be defined for the communication between subsystems. Such a protocol indicates how the communication is done in the several phases. Also, the protocol has to assure that the behaviour of the whole circuit is according to the functional description. One may compare these protocols with the protocol needed for the communication of the whole circuit with the outside world.

Protocols may be defined in several ways. It would be nice if the protocol could be defined in the same way as the functional description of the system. In that case, nothing special should be done for generating the physical circuit for the protocol, since it can be dealt with just as the functional description for the circuit itself. Also, the circuit could perform actions depending on the protocol. For instance, a variable may only be altered if the communication behaves properly.

Protocols must always be verified on their correct behaviour. If a protocol does not behave in the way as defined, the whole circuit will perform a behaviour different from the functional description. Also, one could specify a protocol which not always shows the correct behaviour but in almost all cases. This happens, if circumstances, which lead to a wrong behaviour, occur seldomly. However, when these circumstances occur, the system shows an incorrect behaviour. But if certain constraints are applied to the (sub)circuits, these circumstances may never happen and therefore the protocol always yields a correct behaviour. These assumptions should be verified. Therefore, with certain timing constraints for the subcircuits, no protocol at all could give a behaviour according to the functional description of the system. A timing constraint may be seen as the fact that an action must be performed at least $r_1$ seconds and at most $r_2$ seconds after another action.

Thus, protocols and the combination of the protocol and its environment need to be verified. Properties like deadlock and safeness should be investigated.

Deadlock may happen if two actions are waiting for each other. For example, action $A_1$ needs $b$ and holds $a$ which it releases after getting $b$. Action $A_2$ needs $a$ and holds $b$ which it releases after getting $a$. So, when these actions are done simultaneously, action $A_1$ waits for action $A_2$ and action $A_2$ waits for action $A_1$. Thus, nothing happens. Such deadlock may not happen in the communication between subcircuits.

Safeness is defined as the property that messages are not clobbered. So, a message will not be written while the previous message is not read. Otherwise, the receiving circuit would not know the first message which is useless sent in that case.

These two properties, safeness and no deadlock, are the most important properties that a protocol must fulfill. Therefore, these properties have to be verified for the specified protocol in cooperation with the subcircuits.

2.5 Summary

The previous described features and comparison of synchronous systems and self-timed systems are summarised in the following table.
Thus, one may conclude that designing synchronous systems is more profitable than designing self-timed circuits. But large synchronous systems must be divided into subsystems which communicate with each other. Protocols must be defined for this communication and need to be verified to check the correct behaviour of the system.

This verification of the protocol has to be done in the first stages of a silicon compiler. In the first stages, it is decided to divide the whole circuit into parts. The result is a network of interacting subsystems and this interaction must be verified. Since, if at first sight the system may behave different from its functional description, nothing further should be done. If this verification gives timing constraints at which the system will behave correctly, these constraints must be used in the hardware synthesis phase of the silicon compiler. This hardware synthesis phase must therefore be done later than the verification of the interaction of systems.
3. Demand graphs and petri nets

The purpose of a silicon compiler is to generate a physical circuit from a functional description. An overview of the silicon compiler which will be developed in this department is shown in the figure 3.1. [Jess87a], [Jess87b]. This silicon compiler must be able to convert any functional description into a synchronous circuit for which the layout is generated.

The first stage of the silicon compiler in this figure is the functional description. In principal, the functional description may be in any format and language which suits the purpose of describing a circuit in a functional way. This functional description is parsed and the result is an abstract syntax tree. This syntax tree is well defined and reflects the functional description in a uniform way, no matter what input language was chosen. Therefore, choosing a language for describing the functional behaviour of the system implies that a parser must exist for this language to convert the description.
to the abstract syntax tree.

The major data structure in the next stages of the silicon compiler is the demand graph. With this demand graph, the hardware synthesis phase is done. The demand graph is a kind of data flow graph. Each node in the demand graph corresponds to an operation. Each node has outgoing edges to point to nodes from which it demands data. The incoming edges of a node represent the demands for data from this node. Therefore, the data flow in the demand graph is opposite to the direction of the edges in the demand graph. The abstract syntax tree must be converted to a demand graph and this is done by means of the demand graph constructor.

After constructing the demand graph, it is optimised in several ways. For instance, if some operations are never used, there will be no need to implement these operations in the hardware.

With the demand graph, the hardware synthesis is done. In this hardware synthesis phase, the nodes of the demand graph are mapped onto hardware modules which may perform this operation. This mapping may be done in several ways, for example each operation is mapped onto its own module and will therefore be used only once. On the other hand, all nodes of the demand graph may also be mapped onto just one module which can perform all operations. This module is always used and a controller is needed for scheduling the tasks to perform by this module and to provide the proper control signals to the module. Also, a controller is needed when more than one module is used for the implementation of operators. The controller takes care of the stimuli of the modules and indicates the allocation of the data for the modules.

The hardware synthesis phase is controlled by so-called cost functions. The generated hardware will be the cheapest solution of all possible solutions for mapping the demand graph onto hardware according to the cost functions. Also, constraints may be taken into account. For instance, if an operation may have a maximal delay of \( T \) seconds, the module must perform this action in less than \( T \) seconds even if it is not the cheapest solution according to the cost functions.

The hardware synthesis phase results in a finite state machine description, which describes the necessary operators and the controller. The controller is the implementation of the task scheduler and gives the stimuli to perform an action on some data. The layout of the circuit is generated from this finite state machine description by the layout generator.

The only block which remains in the figure, is the verification. The verification means that a demand graph or a set of interacting demand graphs is checked for exhibiting a correct behaviour. It may give certain constraints to the demand graph. The hardware synthesis phase of the silicon compiler should deal with these constraints to yield a circuit which behaves properly. This verification and the need for it will be discussed in more detail.

### 3.1 Asynchronous elements of the demand graph

As mentioned before, a system will in general be divided in subsystems. Each subsystem is a synchronous circuit with a local clock independent of the clocks in other subsystems. The communication between the subsystems will be asynchronous and the specified protocols have to be verified for the communication between these subcircuits.

Since the demand graph was developed to design synchronous circuits, it has no features to model asynchronous communication. However, the demand graph needs only a slight change to be able to model asynchronous systems also. Waiting for the right moment is an important feature, since in asynchronous circuits an element performs its operation only when it is supposed to do, i.e. when all its needed input data is
ready. One does not have to take care of this waiting in self-timed circuit since it is implicit in the elements, but one has to take care of it in synchronous circuits. Therefore, the syntax tree is extended with an extra base statement: the **on-statement**. This statement has the following syntax

```
( on <test> <body> )
```

*<test>* may be any boolean expression, while *<body>* may be any expression or sequence of expressions. An on-statement indicates when its body may be executed and waits until its condition, i.e. the test, becomes true.

Since asynchronous circuits are only needed for the communication between synchronous circuits, the test of the on-statement must include input operations.

No need for extending the base types of the demand graph nodes exists, since the on-statement may be modelled with the already existing primitives. In principle, this on-statement is a loop in which the body is empty. When the loop is exited, the body of the on-statement may be performed. Therefore, the implementation of an on-statement in the demand graph is about the same as a while loop [Stok86]. The difference is shown in figure 3.2.

\[
\text{value} \quad (\text{while} \ <\text{test}> \ <\text{body}>) \quad (\text{on} \ <\text{test}> \ <\text{body}>)
\]

\[
\text{a)} \quad \text{b)}
\]

**Figure 3.2.** a) a while loop b) a on statement or a waiting loop

The implemented test is the inverse of the test in the on-statement, since the statement *(on <test> <body>)* may be seen as *(while (not <test>) <nothing>)* and after the loop, do *<body>*. However, there is a slight difference between implementing while loops and on statements. This difference is concerned with variables not defined in the loop body. For a while loop, these variables do not demand data from the loop and therefore are not connected to exit nodes of the loop [Stok86]. However, for the on-statement this is necessary, since the action must be delayed till after the positive completion of the test. This yields for all variables defined in the on-body. Each variable defined in the on body must therefore have an exit node to point to. The need for this constraint may be shown by the following example.
(on (and ready (get port-1 acknowledge))
  ((:= request 1)
   (put port-2 request))
)

The demand graph for this statement is shown in figure 3.3.\footnote{All entry and exit nodes have control edges to the control node, i.e. the 'not' operator}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure3.3}
\caption{The demand graph of the example}
\end{figure}

In this case, request may not be set at the output before a previous acknowledge has been given and the action is ready, otherwise the request could already have been sent and the receiver could therefore have got data which was not available. So, the request may be processed only after the test has become true.

3.2 Petri nets

Systems may be modelled by petri nets. A model for a system is used to analyse the system by analysing the model. Petri nets have been proven very useful in the analysis of communication protocols, since properties like deadlock and safeness are very well defined for petri nets. Also, these properties can be easily extracted from the analysis results of the petri net.
If a system may not have deadlocks and must be safe, it would be very nice if such a system can be modelled as a petri net. In that case, the verification of these properties can be done with petri net analysis methods. The same applies to systems which are divided in subsystems which communicate with each other.

A petri net contains two sets of basic elements: transitions and places. A transition represents an event in the net, while places represent conditions. A transition may fire or being executed if certain conditions are fulfilled. These pre-conditions of an event are represented by the input places of the transition. The output places of a transition represent the post-conditions of the transition.

3.3 Mapping of demand graphs onto petri nets

In many cases, a system will be designed as a composition of interacting subsystems. The silicon compiler will generate synchronous circuits which communicate asynchronously with each other. This asynchronous communication has to be verified. This verification is necessary to determine if the generated circuit has the same behaviour as the specified functional description of the system. The verification may also give constraints which guarantee that the generated circuit will behave according to its functional description.

The silicon compiler generates demand graphs as a representation of the system. Since the interaction of systems, i.e. demand graphs, must be verified and since petri nets are well suited for the analysis of interaction, demand graphs should be converted to petri nets. Then, such a petri net can be checked to have certain properties which are also valid for the system which it represents.

The following needs to be defined in the mapping of a set of demands graph to a petri net. The set of demand graphs consists of a demand graph for each subsystem.

- the mapping of nodes of a demand graph to transitions in a petri net
- the mapping of edges of a demand graph to places in a petri net
- the mapping of the communication of the interacting demand graphs to a petri net.

These mappings are discussed in the following paragraphs.

3.3.1 Mapping of a demand graph to a petri net

A demand graph may also be seen as a petri net, since a node of the demand graph, i.e. an operation, may be executed whenever the nodes from which this node demands data, are ready. The demanded data can be seen as being available in the edges of the demand graph. Also, after the execution of a node, other nodes, which demand from this data are possibly enabled to execute. Since nodes in a demand graph and transitions in a petri net perform an action on data, nodes of the demand graph can be one to one mapped onto transitions in the corresponding petri net. Data is mapped onto transitions in the corresponding petri net. Data is represented in the edges of a demand graph and in the places of a petri net. Therefore, edges of a demand graph can be one to one mapped onto places in the corresponding petri net.

Two types of nodes exist in the demand graph:

- normal nodes
  A normal node may be implemented in the hardware whenever all its previous nodes have been implemented. In that case, the data generated by the previous nodes can be processed by this node. The result of this processing is new data which may be needed by other nodes in the demand graph.

- control nodes
  Control nodes do not have the same data transfer mechanism as the normal nodes. These control nodes pass the data from some previous nodes to some outputs. Which
nodes and outputs are chosen for passing the data is determined by the value on the control edge of the control node.

The same distinction exists for types of transitions in a petri net:

- **normal transitions**
  Normal transitions pass tokens from all the input places of the transition to all the output places.

- **control transitions**
  Control transitions pass tokens from some input places of the transition to some output places. Which places are chosen for passing the tokens is determined by the value of the control place of the control transition [Noe73].

Since the petri net must reflect the same behaviour as the demand graph which it represents, normal nodes of the demand graph are mapped on normal transitions of the corresponding petri net. Also, the control nodes are mapped onto control transitions in the petri net.

Here follows an example of the behaviour of a control node. A merge node has two input edges: the value edge and the control edge. It has N output edges, one for each branch in the case statement the merge node stands for. Dependent on the value of the control edge, one of the N output edges is chosen to output the data from the value edge. In the petri net, the merge transition has two input places, the value and the control place and N output places. The transition is enabled to fire if data is available in both the input places. The data of the value place is transported to one of the N output places indicated by the value of the control place.

### 3.3.2 Mapping of the communication between demand graphs onto a petri net

Since the verification will be done by means of petri net analysis methods and the demand graph can easily be converted to a petri net, the communication parts of the circuit have to be modelled also as petri nets.

For verification of protocols it is necessary to know which subsystems communicate with each other. Thus, the connection scheme must be known. Since the system is divided in subsystems, several functional descriptions and demand graphs exist. Each subsystem has its own functional description and demand graph. A demand graph communicates with its environment by means of ports. Each input and output operation indicates which port of the circuit is involved in it. Therefore, the connection scheme has to indicate which ports of a demand graph are connected to what ports of another demand graph and what the direction is of this communication.

Ports of the demand graphs are places in the petri net, while the connections between the demand graphs are represented by transitions. Therefore, the whole system may be seen as a single petri net which can be verified with petri net analysis methods.

A connection may be represented by a single transition in a petri net since a connection performs some action enabled by a pre-condition. The action is to transport data from one side of the wire to the other side. The pre-condition for the data transport is the availability of data at one side of the wire. The post-condition is the availability of data at the other side of the connection. This is precisely the same as the execution rule of a transition in a petri net.

This communication of systems is made visual in figure 3.4. In general, a connection between circuits is made by a wire. However, the connection may also be a more elaborate protocol which have to be modelled in a larger petri net than a single transition. Since the behaviour of such a protocol may also be written in the same way as a functional description, the petri net representation of a connection can be made easily. The result is one large petri net for all the interacting systems.
4. Petri nets

Petri nets were invented as modelling tools by C.A. Petri in 1962 [Petr62]. These nets were developed to be an extension of finite automata. Further refinements to the modelling and analysing power of such petri nets are made by several people. In this chapter, first the basic topics of petri nets are discussed. These topics may also be found in any good book on petri nets, for example the book of J. Peterson [Pete81].

4.1 Mathematical definition of a petri net

A petri net consists of four parts: a set of places, a set of transitions, an input function and an output function. The input function and the output function give a relation between the places and the transitions. The input function indicates which places are the input places of a transition, while the output function indicates what the output places of a transition are.

More formally, a petri net $C$ is a four tuple $(P, T, I, O)$ with $P = \{p_1, p_2, \ldots, p_m\}$ the set of places, $T = \{t_1, t_2, \ldots, t_n\}$ the set of transitions, $I: T \rightarrow P^*$ a mapping from transitions to bags of places and $O: T \rightarrow P^*$ also a mapping from transitions to bags of places.

A place $p_i$ is an input place of transition $t_j$ if $p_i$ is an element of $I(t_j)$. A place $p_i$ is an output place of a transition $t_j$ if $p_i$ is an element of $O(t_j)$. The input and output functions map transitions onto bags of places instead of sets of places, since a place may be a multiple input or output of a transition.

The input and output functions may also be seen as mappings from places to bags of transitions. In this case, a transition is an output of a place if this place is an input place of the transition. A transition is an input of a place if it is an output place of the transition.

4.2 Graphical representation of a petri net

A petri net may also be represented in a graphical way, just like an ordinary graph can. Transitions are represented by bars (|), while circles represent the places. Input places of a transition are indicated by an arrow from these places to that transition. Arrows from a transition to places indicate that these places are the output places of the transition.

Therefore, a petri net may be seen as a bipartite directed graph $G = (V, W, E)$ in which $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$ are the sets of vertices of graph $G$ and $E = \{e_1, \ldots, e_k\}$ is the set of directed arcs. $V$ and $W$ are disjoint sets of vertices. Each arc in $E$ is directing from a vertex in $V$ to a vertex in $W$, or from a vertex in $W$ to a vertex in $V$. So $e_i = (v_j, w_k)$ or $e_i = (w_k, v_j)$.

For a petri net, the two sets of vertices are the set of places $P$ and the set of transitions $T$. Arcs $e_i = (t_j, p_k)$ indicate output places $p_k$ of transition $t_j$. Arcs $e_i = (p_k, t_j)$ indicate the input places $p_k$ of transition $t_j$.

An example of the descriptive and the graphical representation of a petri net is given in figure 4.1.

4.3 Definitions for the behaviour of a petri net

A petri net becomes a powerful analysis tool, when two other definitions are made: markings and execution rule. A marking is an assignment of tokens to the places of a petri net. These tokens may flow through the net according to some execution rules. Each transition has an execution rule, which indicates how tokens are moved from the input places of the transition to the output places.
4.3.1 Marking of a petri net

Mathematically formulated, a marking $\mu$ of a petri net $C = (P,T,I,O)$ is a mapping from the set of places to the non-negative integers $N$, $\mu : P \rightarrow N$. However, a marking is most commonly represented as a vector $\mu = [\mu_1, ..., \mu_m]$ in which each $\mu_i$ is a non-negative integer, which indicates the number of tokens in place $p_i$.

A marked petri net may therefore be seen as the five tuple $(P,T,I,O,\mu)$ with $\mu$ the marking function.

In the graphical representation of the petri net, a marking is indicated by putting tokens in the circles representing the places of the net. Each place $p_i$ gets as many tokens as $\mu_i$.

4.3.2 Execution rule of a transition in a petri net

The execution rule of a transition indicates whether the transition may fire or not. Firing of a transition will result in tokens that are removed from the input places of the transition and inserted in the output places of the transition. A transition may fire whenever it is enabled. The execution rule indicates when a transition is enabled and how the tokens are removed from the input places and inserted in the output places. In general, a transition is enabled by a marking $\mu$ if $\mu(p_i) \geq \#(p_i,I(t_j))$. $\#(p_i,I(t_j))$ is the multiplicity of input place $p_i$ of transition $t_j$, i.e. it is the amount of arcs between input place $p_i$ of transition $t_j$. Therefore, a transition is enabled to fire if each input place has the exact number of tokens or more than it has arcs to the transition.

Firing a transition $t_j$ results in removing a number of $\#(p_i,I(t_j))$ tokens of the input place $p_i$ and inserting a number of $\#(p_{k},O(t_j))$ tokens in the output place $p_k$. 

\[ C = (P,T,I,O) \]
\[ P = \{p_1, p_2, p_3, p_4, p_5\} \]
\[ T = \{t_1, t_2, t_3, t_4, t_5\} \]

\[
\begin{align*}
I(t_1) &= \{p_1\} & I(t_2) &= \{p_2, p_3, p_4\} & I(t_3) &= \{p_5\} & I(t_4) &= \{p_4\} \\
O(t_1) &= \{p_2\} & O(t_2) &= \{p_2\} & O(t_3) &= \{p_3, p_4\} & O(t_4) &= \{p_5\}
\end{align*}
\]
\#(p_k, O(t_j)) is the number of arcs from transition \( t_j \) to output place \( p_k \).

Firing a transition \( t_j \) in a petri net with marking \( \mu \) results in the same (structural) petri net but with marking \( \mu' \). This new marking \( \mu' \) is given by

\[
\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))
\]

Firing a transition never results in a negative number of tokens in a place, since a token is never removed from a place \( p_i \) if it is not in that place. A transition may never be fired, when it is not enabled and a transition is only enabled when \( \#(p_i, I(t_j)) \) or more tokens are in the input places. Only \( \#(p_i, O(t_j)) \) tokens are removed from a place. If less tokens are in the input place of a transition, the transition is not enabled and therefore it may not fire and remove the tokens. Therefore, the number of tokens in a place can never be less than zero.

Firing of a transition is shown in figure 4.2. Also the markings before and after firing are shown.

![Figure 4.2](image)

**Figure 4.2.** Firing of a petri net a) before firing transition \( t_1 \) b) after firing transition \( t_1 \)

### 4.3.3 Next-state function

Another aspect of a petri net is its state. The state of a petri net is defined by its marking, since firing a transition results in a change of the marking. The petri net will go into an other state if a transition fires. Therefore, the next-state function \( \delta(\mu, t_j) \) is defined which is a mapping from a state in the state space of the petri net and a transition to an other state in the state space. The next state function is only defined for a marking \( \mu \) of the petri net and a transition \( t_j \) if transition \( t_j \) is enabled by marking \( \mu \). Therefore the following is valid

\[
\delta(\mu, t_j) = \mu'
\]

where

\[
\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))
\]

only if

\[
\mu(p_i) \geq \#(p_i, I(t_j))
\]

for all the input places \( p_i \) of transition \( t_j \).
4.3.4 Firing sequences

From an initial marking $\mu_0$, one may fire an enabled transition $t_j$, which results in another marking $\mu_1$ which enables other transitions. So, one may construct a sequence of markings $(\mu_0, \mu_1, ...)$, i.e. states, by firing a sequence of transitions $(t_j, t_j, ...)$ which are enabled at the moment of firing. The relation between these two sequences is given by the next-state function $\delta(\mu_k, t_{jk}) = \mu_{k+1}$ ($k=0, 1, ...$). These two sequences represent the execution or behaviour of the petri net.

From the sequence of markings and transitions, a marking $\mu'$ is said to be reachable if a firing sequence exists from an initial marking $\mu$ to marking $\mu'$. A marking $\mu'$ is immediately reachable from a marking $\mu$ if a transition $t_j$ exists in the petri net such that $\delta(\mu, t_j) = \mu'$.

4.4 Modelling with petri nets

One may model any system, which has events depending on conditions, as a petri net. Since each event may be modelled in a transition of the petri net and the conditions are represented by places. An event may take place, whenever its pre-conditions are fulfilled. In a petri net, a transition may fire whenever all its input places have the right amount of tokens. An event may occur if its pre-conditions are true. After taking the action of the event, its post-conditions are true. In a petri net, the transition represents the event, the input places the pre-conditions and the output places represent the post-conditions. A condition is true if the corresponding place in the petri net has a token. So, the occurrence of the event is identical to firing a transition by removing the tokens from the input places to the output places of the transition.

Parallelism is an important aspect in systems and this can also be modelled easily in petri nets. Parallelism is introduced into a system, if two or more events may occur simultaneously. These events have to be independent of each other. In a petri net, two or more transitions may be enabled independently of each other.

In a petri net, events need not to be synchronised since the net is in principle asynchronous. If synchronisation is needed, it has to be modelled in the petri net. However, this can be done easily, since synchronisation is also an event with pre-conditions and post-conditions. Since the petri net is asynchronous, time is not defined in a general petri net. The only time measure in a petri net is the sequence of fired transitions, i.e. the sequence of occurrences of events. However, one may model time in a petri net, but this will be discussed later.

One of the most powerful aspects of modelling a system as a petri net, is the capability of a petri net to model non-determinism. Non-determinism is involved when more than one transition is enabled to fire but the ordering of firing of these transitions is not determined. So each enabled transition may fire but it is not sure that it will fire. One may give precedence rules for firing transitions, but in most cases the determination of the transition that will fire is done randomly.

Non-determinism shows also an other aspect of the behaviour of a petri net: the conflict situation. It may be possible that more than one transition is enabled to fire but firing one transition disables the other transition. Such a case is described as a conflict.

Parallelism or concurrency and the conflict situation are shown in the figures 4.3. and 4.4.
When petri nets were first developed as modelling tools, the execution rule of transitions was not the same as the previously mentioned. The old rule is as follows:

A transition is enabled to fire whenever its input places have the right number of tokens and the output places of the transition have no tokens at all. Tokens are removed from the input places and are inserted in the output places of the transition.

The execution rule of a transition which is nowadays mostly taken is:

A transition is enabled to fire whenever its input places have the right number of tokens. Tokens are removed from the input places and are inserted in the output places of the transition.

The removal of tokens in the input places and insertion of these tokens in the output places of the transition is the same in both execution rules.

For modelling systems, the old method of enabledness of transitions is not a very pleasant one, since in most cases it does not model the behaviour of a system correctly. Most systems will initiate their action if their input conditions are true and nothing takes care of the output conditions. Therefore, enabledness of a transitions has nothing to do with or with not having tokens in the output places of a transition.

However, the old execution rule of a transition can be modelled in the execution rule previously given. This may be seen that a certain condition, which is in fact a post-condition, also is a pre-condition of the transition which must be fulfilled. Therefore, the output places are in some way also fed back to (extra) input places of the transition.

The most easy way to give a post-condition as a pre-condition is the introduction of a so-called inhibitor arc. An inhibitor arc indicates that the place, from which it points to the transition, may have no tokens for enabling the transition. If this input place of
the transition has one or more tokens, the transition is not enabled to fire. So, when a transition may not fire if its output places have one or more tokens, one defines inhibitor arcs from the output places to the transition. Therefore, the output places are also input places of the transition. Such inhibitor arcs are drawn in figure 4.5.

\[ \text{Figure 4.5. A transition which may not fire if tokens in the output places exist} \]

4.5 Analysis of petri nets

Definitions of petri nets are made and the modelling of systems with petri nets is described before. Models of systems are made to get information about the behaviour of a system. Therefore, the model has to be analysed. This means that certain properties of the model, which is our case a petri net, must be discovered. One may consider several properties of a petri net for getting better understanding of the system modelled by the petri net. The following properties are of major concern for petri nets:

boundedness

A petri net is said to be bounded if the number of tokens in each place of the petri net is bounded. This must be valid for all possible states that the petri net can be in. So, there may be never an unbounded number of tokens in a place. A petri net is said to be \( k \)-bounded when the bound of the amount of tokens in a place is \( k \). The special case of 1-boundedness is called safeness.

conservation

A petri net is conservative if the amount of tokens in the net is constant relative to all possible states the net may be in. This implies the fact that each transition has as many output places as it has input places, since only in that case the amount of removed tokens in the input places is equal to the amount of tokens inserted in the output places of the transition. Therefore, a more general definition of conservation of tokens includes a weight factor for each place.

liveness

Several definitions of liveness exist, but the principle thought of liveness is the fact that a transition is live if it may be fired. This does not mean that the transition is enabled, but it can be enabled. A general definition of liveness is:

\[ A \text{ transition } t_j \text{ is live if a possible marking exists in which transition } t_j \text{ is enabled to fire. } \]

The opposite of a live transition is a dead transition. A dead transition is a transition which is never enabled to fire. A petri net is live if all its transitions are live. For the total behaviour of the petri net, each transition is necessary and may therefore not be removed out of the net.

reachability

The reachability problem may be described as:

\[ \text{Is marking } \mu' \text{ reachable from marking } \mu? \]
So, a marking $\mu'$ is said to be reachable from marking $\mu$ if a firing sequence exists which causes marking $\mu$ to transform to marking $\mu'$.

**Coverability**

The coverability problem may be described as:

**Does a marking $\mu'$ exist, which is reachable from marking $\mu$ and has in each place the same amount or more tokens than marking $\mu$?**

**Firing Sequences**

Instead of describing the behaviour of a petri net as a sequence of markings, it may also be described as a sequence of transitions which have fired in that order.

Liveness and firing sequences are closely related, since the liveness of a transition may be determined by investigating the firing sequences of the net. However, reachability is the most important aspect of a petri net, since all other aspects are related to reachability. For example, if marking $\mu'$ is reachable from marking $\mu$, then a firing sequence exists which transforms marking $\mu$ into marking $\mu'$. Each transition in this firing sequence is live. Boundedness, safeness, conservation and coverability of a petri net may be investigated by examining these properties for all the reachable markings of the net.

Liveness is an important property, since it is in most case not useful to design a system which exhibits deadlock. Deadlock between two events occur if one event needs the conditioned established by the other transition and the other way around. The result will be that both these events are waiting forever. Thus, the system will do nothing when a deadlock occurs.

Since the reachability of a marking and the possible firing sequences are the most important aspects in the behaviour of a petri net, these aspects are the directly determined properties. Two major techniques for analysing a petri net are developed:

— analysis tools based on the reachability tree [Pete81]

The reachability tree may be described as the set of all possible markings by firing transition sequences from an initial marking.

— analysis tools based on matrix equations [Mura77]

For a petri net, two matrices, $D^-$ and $D^+$ can be made to represent respectively the input function and the output function.

### 4.5.1 Matrix Analysis

In the matrix analysis, each matrix consists of $n$ rows, representing the transitions, and $m$ columns, representing the places. In fact, this is a way of describing the next-state function, since

$$\delta(\mu, t_j) = \mu + e[j](-D^- + D^+) = \mu + e[j]D$$

with $e[j]$ the unit vector representing transition $t_j$. From this it follows that for a firing sequence $(t_{j1}, \ldots, t_{jk})$:

$$\mu' = \mu + f(\sigma)D \quad \text{with} \quad f(\sigma) = e[j_{j1}] + \ldots + e[j_{jk}]$$

The $j^{th}$ element of the vector $f(\sigma)$ indicates how many times transition $t_j$ fires in this sequence. The reachability problem in this case may be formulated as: Does a solution exist for the above equation when $\mu'$ and $\mu$ are given for the petri net with matrix $D$? Of course, the solution must have non negative integer values.

This matrix approach looks like a very favourable tool for analysing petri nets. However, several serious problems and shortcomings exist. One problem is the fact that the
solution may not be unique but that more solutions exist. This is due to lack of information in the matrix equations, since the matrix \( D \) does not really represent the petri net properly. The matrix \( D \) indicates the total sum of tokens left in a place, but does not distinguish the amount of removed and inserted tokens separately. In case of a self-loop, i.e. an input place is also an output place of the transition, the entry in \( D \) matrix is equal to zero! This reduces the information contained in the matrix \( D \).

Also, the solution of the equation gives only information on how many times each transition has fired, but no information on the order of transition firings. It may also be the case, that a solution is found but no really existing firing sequence can be found.

Because of these problems, the analysis method based on matrix equations in not very helpful and will therefore not be used.

4.5.2 Reachability tree

Since the reachability tree of a petri net with an initial marking represents the reachability set, all possible markings which may be reached from the initial marking have to be computed. In this reachability tree, the firing sequences of the transitions are also recorded, so all described properties of a petri net may be investigated by examining the reachability tree.

First, the visual appearance of the reachability tree in this report will be explained. An example is shown in the second part of figure 4.5. The reachability tree is a method to order all the possible markings of a petri net. This is done in such a way that each node in the tree is a marking, i.e. a state, of the petri net. An edge from a node, which corresponds to marking \( \mu \), to another node, corresponding to marking \( \mu' \), indicates that marking \( \mu' \) can be reached immediately from marking \( \mu \). This may happen if transition \( t_i \) fires at marking \( \mu \). This transition \( t_i \) is the label of the edge.

The reachability tree can be computed by examining all the enabled transitions. For each enabled transition, the marking is computed when this transition should fire. This new marking enables transitions, which are dealt with identically to the previous transition. So, all firing sequences are examined and all reachable markings are computed. However, does this computing of reachable markings ever finish? This means, is the set of reachable markings finite? If the set of reachable markings is infinite, the reachability tree will also have an infinite size. However, a petri net with a finite set of reachable markings may also have an infinite reachability tree if the tree is computed as described before. This is shown in figure 4.6.

It is clear that the reachability tree should be finite, otherwise it can never be computed. This has to be the case as well as if the reachability set of the petri net is finite as if it is infinite. One should find a method of limiting new markings after firing a transition. The first reduction of this amount of new markings occurs if a marking enables no transition. So, the marking is dead. In this case, no new marking can be computed and the reachability tree halts at this marking.

An other reduction of computing new markings is established by duplicate markings. If a marking is the same as a marking previously computed and dealt with, it is not necessary to deal with this marking again. No new information will be available by examining and firing the enabled transitions for this marking.

Another useful reduction is accomplished by markings which cover a marking on the path from the initial marking, i.e. the root of the reachability tree, to these markings. If marking \( \mu' \) covers marking \( \mu \), i.e. \( \mu' \) has in each place an equal number of tokens or more tokens than \( \mu \), and \( \mu' \) is reachable from \( \mu \) with a firing sequence \( \sigma \), this firing sequence \( \sigma \) may be applied to marking \( \mu' \) resulting in marking \( \mu'' \). The result of firing sequence \( \sigma \) on \( \mu' \) was to add \( \mu' - \mu \) tokens to \( \mu \). The same firing sequence will
therefore also add \(u' - u\) tokens to marking \(m'\). This firing sequence \(\sigma^-\) may be applied infinitely often. So, the places, for which marking \(m'\) covers marking \(m\), may get an infinite amount of tokens. All these different markings are not computed, but in marking \(m'\) it is indicated that the places may contain an infinite number of tokens. The infinite number of tokens will be indicated by the symbol \(w\), for which we define

\[
\begin{align*}
w + a &= w \\
w - a &= w \\
a < w \\
w \leq w
\end{align*}
\]

However, some information about the reachability tree is lost by introducing this symbol \(w\), since the exact value of \(w\) is not known. However, it is necessary to bound the size of the reachability tree.

The following algorithm is used for constructing the reachability tree of a petri net with an initial marking. Each node of the tree will be given a type. A node may be of a terminal type, a duplicate type or an interior type. A terminal marking is a marking which enables no transitions. A duplicate marking is a marking already dealt with and will therefore not need examination. An interior marking is marking not occurred before and it has enabled transitions. Frontier markings are the markings which are not given any type yet, and which have to be examined.
Algorithm 1 Construction of the reachability tree

frontier-markings := \{initial-marking\};
while frontier-markings not empty
  \mu_x := choose a frontier marking;
  \text{frontier-markings} := \text{frontier-markings} \setminus \{\mu_x\};
  \text{if} \ \mu_x = \mu_y \text{ for a } \mu_y \text{ in reachability-tree}
  \text{then}
    \mu_x.type := \text{duplicate} (\text{no further processing})
  \text{else}
    \text{if no transitions enabled for } \mu_x
    \text{then}
      \mu_x.type := \text{terminal} (\text{no further processing})
    \text{else}
      \text{for all transition } t_j \text{ enabled by } \mu_x \text{ do}
        \text{create new marking } \mu_z \text{ with for each place } p_i
        \text{case}
        \mu_x[i] = w:
          \mu_z[i] = w
          \text{a marking } \mu_y \text{ on the path from the root of the}
          \text{reachability tree to } \mu_x \text{ exist with } \mu_y < \delta(\mu_x, t_j)
          \text{and } \mu_y[i] < \delta(\mu_x, t_j)_i:
          \mu_z[i] = w
        \text{otherwise:}
          \mu_z[i] = \delta(\mu_x, t_j)_i
        \text{endcase}
        \mu_x.type := \text{interior};
        \text{frontier-markings} := \text{frontier-markings} \cup \{\mu_z\}
        \text{arcs} := \text{arcs} \cup \{(\mu_x, \mu_z)\}
        (\mu_x, \mu_z).label := (t_j)
      \text{endfor}
    \text{endif}
  \text{endif}
endwhile

The algorithm must construct a finite reachability tree, so it has to terminate. This
means that the algorithm cannot always generate new frontier nodes.

The proof of constructing a finite reachability with the given algorithm is done by
contradiction but first needs three lemmas [Pete81].

Lemma 1: In any infinite directed tree in which each node has
only a finite number of direct successors, an infinite path from the
root exists.

Proof: Start at the root with node \(x_0\). Only a finite number of successors of \(x_0\) exist,
but the tree is infinite, so at least one of the successors of \(x_0\) is the root of an infinite
subtree. Since the same reasoning is valid for this successor, an infinite path from the
root of the tree may be constructed.
Q.E.D.

Lemma 2: Every infinite sequence of non-negative integers con­
tains an infinite non-decreasing subsequence.

Proof: Two cases exist:
1. If any element of the sequence occurs infinitely often, the infinite sequence only containing this element is a non-decreasing subsequence.

2. If no element occurs infinitely often, each element occurs a finite number of times. \(x_0\) is an element of the sequence. \(x_0\) has at most \(x_0\) non-negative integers less than \(x_0\) with each of these occurring a finite number of times. So, by continuing far enough in the sequence, an element \(x_1 \geq x_0\) has to be found. Also an element \(x_2 \geq x_1\) can be found. Therefore, an infinite non-decreasing subsequence exist in the sequence.

An infinite subsequence exists in both cases.
Q.E.D.

**Lemma 3:** Every infinite sequence of \(n\)-vectors over the extended non-negative integers, i.e. \((0,1,...) + \{w\}\), contains an infinite non-decreasing subsequence.¹

**Proof:** By induction on \(n\), the dimension of the vector space.

1. Base (\(n=1\)). If an infinite number of \(w\) vectors exist, they form an infinite non-decreasing subsequence. If not, then the infinite sequence formed by deleting the finite number of \(w\) vectors has an infinite non-decreasing subsequence according to Lemma 2.

2. Induction hypothesis. Consider the first coordinate of the vectors in the sequence. If an infinite many of vectors exist with the first coordinate \(w\), select this infinite subsequence which is non-decreasing in the first coordinate. Otherwise, consider the infinite subsequence of vectors, according to Lemma 2, of the sequence by deleting the finite number of vectors with \(w\) as first coordinate. Thus, an infinite subsequence non-decreasing in the first coordinate exists. With the induction hypothesis that an infinite non-decreasing subsequence exists for vectors with dimension \(n-1\), an infinite non-decreasing subsequence exists for an infinite sequence of vectors with dimension \(n\).
Q.E.D.

Now, we can proof that the reachability tree is finite.

**Proof:** The proof is by contradiction, therefore assume that an infinite reachability tree exists. By Lemma 1, an infinite path from the root of the tree exists since the number of possible successors is limited by the number of transitions. Thus, \(\mu_{x_0}/\mu_{x_1}\ldots\) is an infinite sequence of \(n\)-vectors over the set of extended non-negative integers. By Lemma 3, there is an infinite non-decreasing subsequence \(\mu_{x_1} \leq \mu_{x_2} \leq \ldots\). Because of the fact that duplicate nodes are not processed when the reachability tree is constructed, \(\mu_x \neq \mu_y\). Therefore, the previous given sequence is an infinite increasing sequence. But by constructing the reachability tree, if \(\mu_x < \mu_y\), at least one component of \(\mu_x\), which is not \(w\), will be replaced by \(w\) in \(\mu_y\). Thus, \(\mu_{x_1}\) has at least one \(w\)-component, \(\mu_{x_2}\) has at least two \(w\)-components, etc. Since the markings are \(n\)-dimensional, \(\mu_{x_{in}}\) has all its components \(w\) and \(\mu_{x_{in+1}}\) is not greater \(\mu_{x_{in}}\), the initial assumption is not valid. Therefore, no infinite reachability tree can exist.
Q.E.D.

---

¹ A vector \(\mu_0\) is less than vector \(\mu_1\) if each element of vector \(\mu_0\) is less than the same element in vector \(\mu_1\).
From the algorithm for constructing the reachability tree of a petri net, it may be seen that there is not only one reachability tree of a petri net, but there may be more than one. This is due to the order of processing frontier nodes. If one takes another order, the reachability tree may be different, since for instance some markings are not duplicate in this reachability tree as it is in another reachability tree. This points to the problem that there is no easy way to check if two petri nets have the same behaviour by computing the reachability trees of the two petri nets.

Safeness and boundedness of a petri can be easily checked with the reachability tree. The petri net is bounded if no marking exists with an \( w \)-element. The petri net is safe if all markings have at most one token in each place.

However, not all problems can be solved with the reachability tree due to the symbol \( w \). The symbol \( w \) causes a loss of information, since the individual number of tokens is discarded and only a large amount of tokens is remembered. In some cases, \( w \) may stand for all integers, while in another case \( w \) may stand for odd integers. Therefore, petri nets with the same reachability trees do not need to have the same reachability sets. The reachability tree contains necessary information, but it may be insufficient.

However, for a petri net with a reachability tree containing no markings with \( w \), all problems may be solved with the aid of the reachability tree. Thus, if a petri net under concern must be bounded, this boundedness can be investigated. If the net is bounded, the problems about liveness etc. can also be solved.

Now, the visual appearance in this report of the reachability tree of a petri net will be further explained. An example is shown in figure 4.7. Three types of a state of a petri net must be made clear in the reachability tree:

- **interior node**
  These nodes indicate a state of the petri net at which transitions are enabled to fire. The enabled transitions are those indicated by the labels of the outgoing edges of the node.

- **terminal node**
  This node represents a state at which no transitions are enabled to fire. Therefore, they have no outgoing edges. These nodes are accentuated in the visual appearance of the reachability tree by a double circle.

- **duplicate node**
  Such a node represents a state which has already been processed in the reachability tree. Therefore, these nodes have no outgoing edges. However, in the visual appearance, such a node has one outgoing edge with no label. The node to which this edge points, is exactly the same state of the petri net.

### 4.6 Reductions

Constructing the reachability tree for a given petri net means to compute almost all possible reachable markings, i.e. the state space of the petri net. The result is a lot of computations, since the state space of a petri net grows enormously. One may compare the state space of a petri net with the state space of a finite state machine, especially the non-deterministic finite state machine. Since the modelling power of petri nets is beyond the modelling power of a non-deterministic finite state machine, the state space may even be bigger.

Therefore, the reachability tree construction has to be reduced if possible. If one should want to know all possible markings, the construction of the whole reachability tree is necessary. If only some properties of a petri net need to be known, it would not to be necessary to compute all possible markings but only a subset. If one could predict the behaviour of a reachability subtree by not computing the whole subtree, the whole reachability tree need not to be constructed. The information contents of a subtree of...
the reachability tree may also be extracted from an other subtree of the reachability tree.

4.6.1 Anticipation of subtrees in the reachability tree

The following example shows that subtrees may give the same information and therefore not all subtrees have to be constructed. The petri net and a reachability tree of this net are shown in figure 4.7.

Figure 4.7. a) A petri net b) The reachability tree

It can easily be seen from the reachability tree that the overall behaviour of the petri net of figure 4.7. is to fire transitions $t_1$ and $t_3$ concurrently. The overall behaviour of the petri net is live and bounded. Transitions $t_1$ and $t_3$ are conflict-free transitions, i.e. firing of transition $t_1$ respectively $t_3$ does not disable transition $t_3$ respectively $t_1$ when both transitions are enabled. With this property in mind and the result of the subtree $t_1t_2t_3$ of the reachability tree, the subtrees $t_1t_3t_2$ and $t_3t_1t_2$ can be thought of being also possible.

The same applies for more transitions which may fire independently from each other and also give no rise to conflict situations, for example the net in figure 4.8. The end effect of all possible firing schedules is the same marking. Therefore, only one schedule should have been computed for getting the same conclusion about the behaviour of the petri net. Thus, the reachability can be reduced and one has to prove that (the behaviour of) the subtree in the reachability tree of a petri net with initial marking $p$ and first fired transition $t_j$ is member of the subtree with initial marking $f$ and initial firing sequence $l_1l_j$. Also transition $t_i$ is conflict free.

**Proof: anticipation**

Since transition $t_i$ is a conflict free transition, it is always enabled to fire whenever it is once enabled, at least until it gets fired. Therefore, a firing sequence $\sigma^-$ exists, from the root $\sigma$ and initial sequence $l_j$ of the subtree under concern, till transition $t_i$. This sequence $\sigma^-$ can also be fired in the subtree from the root $\sigma$ with initial sequence $l_1l_j$, since firing of transition $t_i$ disables no other transitions. However, it may occur that firing of transition $t_i$ enables other transitions. Thus two firing sequences are of
These two firing sequences result in the same final marking, since transition $t_i$ is a conflict-free transition. Therefore, the second firing sequence need not to be computed, since it gives no extra information on the behaviour of the petri net.

Q.E.D.

Note that in the proof nothing is said about transition $t_j$ whether it should be conflict-free or not. Therefore, a simple heuristic can be made to reduce the computation of the reachability tree: first compute subtrees with initial fired transition a conflict-free transition. After all the enabled conflict-free transitions are processed, process the enabled conflictive transitions if necessary.

The transition $t_j$ in the proof above given is said to be anticipated by transition $t_i$ at marking $\mu$. It is also clear that transition $t_j$ may be anticipated at marking $\mu$ if it is an anticipated transition at marking $\mu'$.

The end marking may also be computed by adding the initial marking and all the input and output markings of the transitions. In this case, the intermediate markings are not computed but only the final marking. If the overall characteristics of a petri net have to be extracted from the reachability tree, the interior nodes are of no concern. Only the firing sequences, liveness and boundedness of the petri net are important.

One may imagine that concurrent, conflict-free transitions may be fired all at once with no loss of information on the behaviour of the petri net. One should only keep in mind that all orders of firing the transitions may be possible. Not computing all this permutations reduces the (computed) state space a great deal.

For this reduction of the reachability tree, one has to proof that the behaviour of the subtree in the reachability tree of a petri net with initial marking $\mu$ and enabled conflict-free transitions $t_i \ldots t_j$ is identical to the behaviour of the subtree when all these
conflict free transitions are fired at once.

Proof: multiple firing
Since all transitions $t_i..t_j$ are conflict free, a firing sequence $t_i..t_j \sigma^-$ from initial marking $\mu$ exists. Also, firing sequences starting with another initial transition in this conflict free transition sets exist. But such a sequence may be anticipated. Therefore, one may first add all the enabled conflict free transitions and after that construct the rest of the reachability tree since no new information can be extracted from the other sub-trees in the reachability tree.

Q.E.D.

4.6.2 Well-behaved subnets of a petri net

The reachability tree may also contain several common subsequences which occur more than once. If all these subsequences are identical, they are essentially time consuming computations. No extra information can be extracted from these identical subsequences. Therefore, one could examine such a sequence once and never again. In principle, this means that some transitions may be mapped onto one transition. The mapped transitions form also a petri net which may be examined. But in the total petri net, this set of transitions are represented by one transition. Whenever this set of transitions would be enabled to fire, i.e. if a particular fixed firing sequence would occur, in the original petri net, one should fire only this transition in the reduced transition. The input places of the set of transitions in the original petri net are all input places of the transition in the reduced net. The same applies to the output places of the set of transitions in the original net.

A petri net may also be seen as a composition of several subnets. Each subnet is also a petri net and the total construction of subnets is again a petri net.

If each subnet is a so-called well-behaved net and the petri net with each subnet replaced by one transition is also well-behaved, then the whole petri net is well-behaved. A petri net is well-behaved if it is safe and it is life [List85a], [List85b]. It is clear that only the behaviour of such a petri net can be analysed in which only the properties safeness and liveness are necessary.

Each subnet communicates to its environment by its input and output places, so a subnet may be seen as a single transition with input and output places. The proof of this concept of well-behaved petri nets is as follows:

Proof: The following cases exist:

— If the petri net with each subnet replaced by a single transition is live, then a firing sequence exists in which this transition is enabled to fire. Therefore, if this subnet is not safe and/or live, then the total petri net is also not safe and/or live.

— If the petri net with each subnet replaced by a single transition is not live, then the whole petri net will also not be live since transitions exist which never get enabled to fire.

— If the petri net with each subnet replaced by a single transition is not safe, then the whole petri net will also not be safe since places exist which can have more than one token.

This concept of well-behaved subnets may reduce the giant enumeration of all possible states of a petri net. It is clear that the smaller the net is, the smaller the state space will be, since the state space is dependent on the number of transitions and places, and especially on parallel transitions which are independent of each other. Also it may be that the sum of analysing several nets is less complex than the analysis of the total petri net alone.
This concept may also be further refined. This means that a subnet may also be divided in several subsystems. If each sub-subsystem is well behaved and the subsystem with each sub-subsystem substituted by one transition is well-behaved, too, then the subsystem is well-behaved. So even smaller petri nets have to be analysed for checking the behaviour of a petri net.

A petri net may therefore be set up hierarchical and analysed also in that way. The analysis and development of a petri net may be done bottom-up or top-down. Top-down means that first the highest level of a petri net is designed and verified, after that the one but highest level is designed and verified. This process carries on until the lowest level of hierarchy is reached. Designing a well-behaved petri net means that each time a well-behaved subnet must be designed in the well-behaved petri net of a higher level. If one of the designed subnets is not well-behaved, the whole petri net will also be not safe and live and one may stop the design process or design another, well-behaved subnet.

Bottom-up verification is done when subnets are discovered in a larger net structure. Such a subnet is than verified. This is done for all subnets. After processing this lowest level, the one but lowest level is processed in the same way. This process is propagated to the highest level of hierarchy.

In principle, this bottom-up strategy is identical to the top-down strategy in designing and verifying a whole petri net. The top-down strategy is mostly used when designing a system, while the bottom-up strategy can be adopted for verification of a system when the system is not designed hierarchically or the hierarchy is not known.

An example of such a reduction in well-behaved subnets is the substitution of a long chain of transitions by one transition as indicated in the figures 4.9. and 4.10.

The two petri nets do not have identical safeness properties.

**Figure 4.9.** Well-behaved petri net with well-behaved subnet

The first example shows a well-behaved petri net, while the second example shows a petri net which is not well-behaved. A well-behaved subnet can be extracted from these two petri nets. It consists of the transition chain \( t_2 t_3 t_4 \) and is identical in both petri nets. It is clear that this subnet is safe and live and therefore well-behaved. The only possible firing sequence, if initially only transition \( t_2 \) is enabled to fire, is \( t_2 t_3 t_4 \). Since each place never has more than one token and each transition gets once enabled,
This subnet can be substituted by the single transition \( t_2 \) in the original petri nets. However, the petri net of figure 4.11. is well-behaved, while the petri net of figure 4.12. is not. The only possible firing sequence in the net of figure 4.11. is \( t_1t_2 \) and no place gets ever more than one token. This net is well-behaved just like its subnet. In the petri net of figure 4.12., transitions \( t_1 \) and \( t_2 \) may always fire. The result is that place \( p_2 \) may contain any number of tokens. Thus this petri net is not well-behaved, while its subnet is.

4.7 Petri nets with time

As mentioned before, time is not involved in general petri nets, since all transitions fire in an instantaneous and non-deterministical way. Only an order of occurred events is a notion of time. However, most systems have a more detailed time notion. Therefore, it would be nice if this concept of time is also supported by petri nets. Otherwise, one would not choose a petri net as the model for examining the behaviour of a system.

A few approaches for inserting time in petri nets are made. These approaches are:

**Timed petri nets**

[Ramc73]

In this petri net, the duration of the firing of a transition is specified. Thus, a transition no more fires instantaneously but during a time interval. Transitions are enabled as in normal petri nets, but the firing of a transition involves some time \( \tau \). So, the firing of a transition can be divided into two steps. First, tokens are removed out of the input places of the transition. Second, tokens are inserted in the output places of the transition a time \( \tau \) later.

Timed petri nets are very useful if the exact delay times of the transitions are known.
Time petri nets
[Merl74]
Now firing intervals are given for each transition, in which the transition fires instantaneously. However, the actual firing of a transition is still instantaneous. The firing interval is not in between the moment of enabling and infinity but in a specified interval. Each transition has an associated minimal time and a maximal time. The transition may fire at any moment between $t + t_{\text{min}}$ and $t + t_{\text{max}}$ with $t$ the time at which the transition is enabled.

Time petri nets are very useful to model variable delay times and time-outs and are therefore very useful in the validation of communication protocols.

Timed place-transition nets
[Sifa79]
In this model for petri nets, time is associated with the places of the net instead of the transitions. Two kinds of tokens are distinguished: available tokens and unavailable tokens. A transition may fire whenever it is enabled when only the available tokens are concerned in the input places. The output places of the firing transition get unavailable tokens. Unavailable tokens become available after a delay $\tau$ of the place in which the token is.

One can prove that the timed place-transition nets are equivalent to timed petri nets, since a timed place-transition net can be transformed to an equivalent timed petri net. A timed petri net can be transformed in a timed place transition petri net, too. These transformations are shown in the figures 4.11. and 4.12.

![Figure 4.11. Transformation of a place of a timed place transition net into a subnet of a timed petri net.](image)

Also, timed petri nets can be modelled as time petri nets, since all transitions of a timed petri net can be modelled as two transitions and one place in the time petri net model as indicated in figure 4.13. Since, also normal petri nets can be seen as a special case of time petri nets, time petri nets are the most general model for petri nets. Normal petri nets may be seen as time petri nets, since for each transition in the normal petri $t_{\text{min}} = 0$ and $t_{\text{max}} = \omega$.

Some comparisons can be made between a time petri net and its corresponding normal net without timing.2
Reachable markings and firing sequences

If a transition may fire in the normal net, it is not sure that it may also fire in the time Petri net because of its timing constraints. Therefore, all possible firing sequences of a time Petri net are a subset of all possible firing sequences of the corresponding normal Petri net. Thus, the reachability set of a time Petri net is also a subset of the reachability set of the corresponding normal Petri net, since if a transition in the time Petri net may fire, it may also fire in the normal net.

2. The corresponding normal Petri net of a time Petri net is the net with the same structure, i.e. the same places and transitions and their interconnections, but transitions have no timing constraints.
States of the petri net
In a petri net without timing constraints, states of the net are the possible markings. However, in a petri net with timing, states are not uniquely identified through the markings anymore. Because of the timing constraints, different states may have the same markings but the next possible markings may be different for these markings. States of a time petri net are formed by the marking and the timing constraints of the transitions.

Boundedness
The reachability set of the time petri net is a subset of the reachability set of the corresponding normal net. Therefore, boundedness of the corresponding normal petri net is sufficient for the boundedness of the time net but not necessary. Also, if the time petri net is bounded, the normal net may be unbounded. However, if the time petri net is unbounded, the normal net is also unbounded.

Liveness
No real relation exists between the liveness of the time petri net and the corresponding normal net.

In time petri nets, each transition has a minimal time and a maximal time. These times define an interval in which the transition fires. The firing of the transition may occur at any time in this interval. The firing is still instantaneous. If a transition \( t_i \) fires, it will enable transitions \( t_j \). These transitions may fire in the interval \( <\tau + t_{\text{min}_j}, \tau + t_{\text{max}_j}> \). \( \tau \) is the absolute time at which transition \( t_i \) is fired. However, this absolute time is not known and therefore the absolute time interval in which transition \( t_j \) will fire is not exactly known.

A time interval may be made, for instance \( <t_{\text{min}_j} + t_{\text{min}_j}, t_{\text{max}_j} + t_{\text{max}_j}> \), in which transition \( t_j \) will fire. Transition \( t_i \) fires at time \( \tau \), and no other transition may fire before this time. Therefore, the time origin of the net may be moved by \( \tau \) seconds. Each bound of the interval in which an enabled transition may fire will be moved this amount of time. The minimal bound may never get less than zero, since the net is examined at time 0. The maximal bound of the time interval of a transition may certainly not become less than 0, since in that case this transition had to be fired before! The intervals for the new enabled transitions \( t_j \) will then be \( <t_{\text{min}_j}, t_{\text{max}_j}> \). The time variable \( \tau \) is removed for these transitions. This variable \( \tau \) may also be removed out of the shifted intervals of the transitions already enabled. This removal will be done according to the time interval of the fired transition. This removal of the time at which transition \( t_i \) fires, may give certain constraints to the intervals of the other enabled transitions.

4.7.1 Example
In the following example, it will be made more clear that the firing of a transition may give timing constraints for the firing interval of other enabled transitions. The petri net of this example is shown in figure 4.14.
Transition \( t_i \) may fire in the interval \( <0,2> \).
Transition \( t_j \) may fire in the interval \( <1,3> \).
Transition \( t_k \) may fire in the interval \( <0,2> \).
If transition \( t_i \) fires, no other transitions get enabled.
When transition \( t_i \) fires at time \( \tau \) (with \( 0 \leq \tau \leq 2 \)), the interval for transition \( t_k \) will be \( <\tau, 2> \). For transition \( t_j \), the new interval will be \( <\tau, 2> \) if \( 0 \leq \tau \leq 1 \) and \( <\tau, 3> \) if \( 1 \leq \tau \leq 2 \). These two different intervals may also be given by the interval \( <\max(1, \tau), 3> \) for \( 0 \leq \tau \leq 2 \).

When the time origin is shifted to \( \tau \), the new interval for transition \( t_j \) will be \( <\max(0, 1-\tau), 3-\tau> \) and for transition \( t_k \) \( <0, 2-\tau> \). However, it would be nice if this time \( \tau \) could be eliminated from the firing intervals, since it has nothing to do with
Figure 4.14. Example of timing constraints caused by the firing of transition $t_i$

the remaining enabled transitions. Therefore, the firing are rewritten:

Transition $t_j$ may fire in the interval $<0,3>$.  
Transition $t_k$ may fire in the interval $<0,2>$.  
Transition $t_k$ must fire earlier than 1 time unit after the time $t_j$ fires. Thus the time interval for transition $t_k$ becomes $<0, \text{min}(2,1+T_{t_j})>$ (with $T_{t_j}$ the time at which transition $t_j$ will fire).

This last rule is an example of a timing constraint caused by the firing of a transition. This constraints is depends only on the time an enabled transition will fire and not on an already fired transition.

4.7.2 State of a time petri net

Firing a transition give constraints for the time intervals of the enabled transitions. Therefore, a state of a time petri net is determined by the marking and all the timing constraints of the enabled transitions. Given a marking and the timing constraints, the transitions which may fire must be computed. A transition $t_i$ may fire if no other transition $t_j$ has to fire before transition $t_i$. This means that a solution of a system of inequalities must be found. The system of inequalities are the timing constraints and may be written as:

$a_i \leq \tau_i \leq b_i$ for all enabled transitions $t_i$  
$\tau_i - \tau_j \leq c_{ij}$ for all pairs of enabled transitions $t_i$ and $t_j$ ($t_i \neq t_j$)

Firable transitions are enabled transitions which may fire according to the timing constraints of the time petri net which are valid at this time. The firable transitions are those for which no other transition must fire earlier. These firable transitions have to be computed from the system of inequalities given before. The problem of solving the timing constraints is less difficult if the variables $a_i$ are the smallest possible values for $\tau_i$, $b_i$ the largest possible values for $\tau_i$ and $c_{ij}$ the largest possible difference of $\tau_i$ and $\tau_j$. Such a system is called a canonic system [Bert83], [Mena85]. The following equation is therefore valid for a canonic system:

$c_{ij} \leq b_i - a_j$

When the system of inequalities is canonic, the solution can be found in a reasonable way. In general, solving a system of inequalities is probably a NP-complete problem. If all system inequalities have no more than two variables, the solution can be found in polynomial time [Aspv79].

The system of inequalities, which represents the timing constraints, is called the domain. The domain $D$ may be represented by two vectors $a$ and $b$ and a matrix $C$. The rows of the two vectors and the matrix represent the enabled transitions.

A state of a time petri net is given by the pair $(\mu, D)$ with $\mu$ a marking and $D$ a domain. Algorithm 1 still returns the reachability of the time petri net if the pairs
\((m, D)\) are examined instead of markings. States of a net are equal if they have the same markings and if the domains are equal. Domains are equal if they have the same solution. Thus, domains may be structural not equal, but are called equal if they yield the same solution.

One should find the solutions of systems of inequalities for determining the equality of two domains. It is less difficult to compare the structure of two domains then to compare the solutions, which have to be computed. Therefore, it would be nice if the domains are represented in such a way that domains have the same solution, only if they have identical structures. In this case, two domains \(D_1\) and \(D_2\) are said to be equal, if

\[
\begin{align*}
a_1 &= a_2 \\
b_1 &= b_2 \\
c_1 &= c_2
\end{align*}
\]

Of course, domains may only be equal if they represent the same enabled transitions. For verifying the equality of the vectors \(a\) and \(b\) and the matrix \(C\), it may be necessary that they are reordered in such a way that each row of the separate domain components represent the same enabled transition.

4.7.3 Execution rule of a transition in a time petri net

In a normal petri net, a transition may fire whenever it is enabled. However, in a time petri net this is not always valid. The execution rule of a transition in a time petri net is a bit more complicated. Two cases have to be distinguished: the marking and the domain. Therefore,

\[
\text{a transition } t_i \text{ in a time petri may fire, if it is enabled and no other transition } t_j \text{ must fire before transition } t_i.
\]

4.7.3.1 Finding the firable transitions

According to the timing constraints, it may occur that a transition must fire before any other transition. For example, if the interval in which enabled transition \(t_i\) may fire is \(<0,1>\) and for transition \(t_j\) it is \(<2,3>\). It is clear that transition \(t_i\) has to fire before transition \(t_j\).

In terms of the domain equations, not firable transitions are those transitions for which the solution of the domain results in an interval which lies after the solution interval of another transition and does not overlap with it.

The firable transitions in a domain are the transitions for which the next equations are valid. However, it is assumed that \(a_i\) is the smallest possible value for \(\tau_i\), \(b_i\) the largest possible value for \(\tau_i\) and \(c_{ij}\) the largest possible difference between \(\tau_i\) and \(\tau_j\). So, the domain must be canonic.

The following set of equations are valid for a transition \(t_i\) which may be fired at this time:
These equations result in the following conditions for a transition $t_i$ which may fire at this moment.

$$a_i \leq t_i \leq b_i$$
$$a_j \leq t_j \leq b_j$$
$$t_j - t_i \leq c_{ji}$$
$$t_i - t_j \leq c_{ij}$$
$$t_j - t_k \leq c_{jk}$$

4.7.3.2 Firing a firable transition

If a firable transition will be fired, both the marking and the domain must be updated. The new marking is computed as in the normal petri net. Thus, tokens are removed from the input places of the fired transition and inserted in the output places of the transition. However, the domain can not be updated so easily. The domain must be updated as follows:

1. Add the equation $t_i \leq t_j$ to the system of inequalities.
2. Eliminate the transitions which are in conflict with transition $t_j$ from the domain.
3. Shift the time origin with $t_i$ seconds to the right.
   This means that all new $t_j$'s are formed by adding $t_i$ to $t_j$.
4. Eliminate $t_i$ from the domain.
5. Add $t_{\min}$ and $t_{\max}$ for all new enabled transitions to the domain.

Step 1 has to be done, since transition $t_i$ fires. Because of transition $t_i$ is a firable transition it may fire before any other transition and if it fires, $t_i$ is less than any other $t_j$. This time $t_i$ has no meaning anymore and has to be eliminated from the system of inequalities. So, all time intervals may be shifted by this amount of time.

Step 2 is necessary, because the transitions which are in conflict with the fired transition are no longer enabled. This elimination yields the following equations for the domain:

$$a_j' = \max (a_j, a_i - c_{ij})$$
$$b_j' = \min (b_j, c_{j1} + b_i)$$
$$c_{jk} = \min (c_{jk}, c_{j1} + c_{ik})$$

These equations can be extracted from the following set of equations which are the conditions when transition $t_i$ must be deleted. This transition $t_i$ is in conflict with transition $t_i$ which has fired.
Shifting the time origin $\tau_i$ seconds left and eliminating the fired transition yields the following domain

$$a_j' = \max (0, a_j - b_i, -c_{ij})$$

$$b_j' = \min (c_{ji}, b_j - a_i)$$

$$c_{jk}' = \min (b_j - a_k, c_{jk}, b_j' - a_k')$$

This set of equations is extracted from the following equations which are valid when transition $t_i$ has fired.

$$\tau_i \leq \tau_j' + \tau_i$$

$$a_i \leq \tau_i \leq b_i$$

$$a_j \leq \tau_j' + \tau_i \leq b_j$$

$$\tau_j' - \tau_k' \leq c_{jk}$$

$$\tau_j' + \tau_i - \tau_i \leq c_{ji}$$

$$\tau_i - \tau_j' - \tau_i \leq c_{ij}$$

The resulting domain of the above given equations is a domain which is canonic. Therefore, the constraints given before yield the transitions which are enabled to fire.
5. Implementation of the verification of systems

As described before, all parts which constitute a circuit can be written in the same functional description. This functional description can be converted into a demand graph. The demand graph constructor has only to be altered slightly to model asynchronous communication. Only an implementation must be given for a wait statement, i.e. the previously mentioned on-statement. This on-statement may be seen as a while statement with empty body and the on-body must be implemented as the exit from the loop.

5.1 Conversion of demand graph into a petri net

The demand graph can be converted to a petri net by mapping each node in the demand graph onto a transition in the corresponding petri net. An output edge of a node in the graph will be mapped onto an input place of the corresponding transition in the petri net. An input edge in the demand graph will be mapped onto an output place in the petri net. This change of input in the demand graph to output in the petri net and output in the graph to input in the net is necessary to indicate the proper data flow through the demand graph and through the petri net. Nodes in the demand graph demand data from nodes, which are pointed by the outgoing edges of this node. Therefore, the data flow in the graph is opposite to the direction of the edges. In a petri net, the data flow is from input places through a transition to the output places. So the direction of tokens in a petri net is in the opposite way of the edges in a demand graph.

The control nodes of the demand graph cannot be mapped onto normal transitions in the petri net. Control nodes have their data flow from some input edges to some output edges. The value of the control edge of the control nodes determines which edges are chosen. These nodes are mapped onto so-called control transitions which represent the same action as the control nodes of the demand graph. These control transitions pass transitions from some input places to some output places depending on the control place of the transition [Noe73]. The control nodes of a demand graph are the nodes which represent switches in the functional description of the system. A switch can be seen as directing a particular input to a particular output depending on a control value. This direction indicates the data flow of the node.

Not only the demand graph edges are mapped onto places in the petri net, the edges of the IO-graph and the ports of the communication nodes are also mapped onto places in the petri net. Nodes of the IO-graph are also nodes of the demand graph and are therefore transitions in the petri net. These nodes are the one which perform the communication with the environment of the system. The edges of the IO-graph indicate in what order the nodes of the IO-graph must be activated. This means that the order indicated in the IO-graph must be maintained in the implementation of the graph. In terms of a petri net, the edges control whether a transition is enabled or not are therefore modelled as places in the petri net. The order in the IO-graph is the same as given by the functional description of the system. Otherwise, the user would not know when to give what data to the system and when what data would be available.

The communication nodes on the demand graph are the put and get nodes. The put node outputs data from its source edge to the environment of the system through its port, while a get node inputs data from the environment to the system through its port.

The communication between systems takes place through connections which are modelled by transitions in the petri net. Data, outside two communicating systems, flows from an output of one system to an input of the other system. Thus, the input places of a connection transition are those places which correspond to a port of a put node in a subsystem. The output places of a connection transition are the ports of the get nodes. The transition which models a connection may also be a larger petri net.
which should model the behaviour of the connection more accurately.

The implementation of the connection scheme of interacting systems is a list. Each element of this list indicates which ports are connected with each other and what the direction of the connection is. Therefore, each element of the list will be modelled as a single transition in the petri net.

Not all nodes of the demand graph can now be converted to the appropriate transitions in the corresponding petri net. The nodes which are necessary for the calling and leaving of procedures and functions in the functional description of a system [Stok86], are not converted to transitions in the petri net. Procedures and functions may be seen as subparts of the system, which have a behaviour independent of the rest of the system. Therefore, they may be designed as separate systems, which communicate with the main system through interface nodes. These interface nodes indicate which procedure or function is called and by whom. This last information is necessary to give the output of the procedure or function to the appropriate part of the system. Also, a procedure or function may be called only by one main body at a time and it must return the right values to the proper places.

So, it is clear that procedures and functions act like interacting systems and that they must fulfill the same properties as an ordinary communication protocol. Therefore, they should be modelled in the same way. This means that interface nodes of procedures and functions must be modelled in the petri net as put and get nodes of a demand graph. Therefore, the interface nodes of procedures and functions are not modelled just like normal transitions in the petri net but have to be modelled as communication transitions.

5.2 Analysis of the petri net

A petri net consists of a set of transitions and a set of places. Each transition has an input vector and an output vector. The input vector indicates which places of the set of places are the input places of the transition. The output vector indicates the output places of the transition. The vectors can directly be used for computing the next state of the petri net, when a transition fires. If the transition is a control transition, the input and/or output vector may be a vector of such vectors. The value of the control place determines which input and output vector are chosen for the firing of such a control transition. The transitions have a type to distinguish, for example, control transitions from normal transitions.

The resulting petri net can be analysed by petri net analysis methods. These analysis methods are based on the reachability set of the petri net. Therefore, a reachability tree for the petri net should be constructed first. The reachability tree of a petri net is constructed with Algorithm 1 in chapter . A recursive approach is chosen for implementing the construction of the reachability tree. Therefore, the reachability is built in a depth first search. Thus, when more than one transition may fire at a marking, first a subtree is dealt with. This means that a reachability tree is constructed with the current marking as initial marking. The construction of a subtree is identical to the construction of the whole tree. The subtree starts with the firing of an enabled transition. When the complete subtree has been made, the next subtree is made. This continues until all enabled transition are dealt with.

The reachability tree consists of nodes. Each node represents a state of the petri net. A node in the reachability tree is a structure with the following fields:

- from-marking
  This field contains the node from which this node was reached.

- to-marking
  This field contains the nodes which may be reached from this node.
This field contains the type of the node. A node is a terminal node if no other nodes can be reached from this node. The node is a duplicate node if it is already processed in another part of the reachability tree. Otherwise, the node is an interior node of the reachability tree.

**duplicate-of**
This field of a node does only exist if it is a duplicate node. In that case, it contains the node of which this node is the duplicate.

**transition**
This field contains the transition which must be fired to reach this node of the reachability tree from the previous node. So, it is the label of the arc between two nodes.

**domain**
This field contains the domain of a state of the petri net. This domain is a vector with four elements:

- **a-vector**
  This vector contains the smallest possible values for the timing constraints of the possibly enabled transitions.

- **b-vector**
  This vector contains the largest possible values for the timing constraints of the possibly enabled transitions.

- **c-matrix**
  This matrix contains the largest possible differences for the timing constraints of each pair of possibly enabled transitions.

- **transition-list**
  This list contains the possibly firable transitions of this state.

An initial marking is needed to construct the reachability tree. This initial marking is derived from the demand graphs of which this petri net is the representation. One can easily find the nodes in the demand graph which have to give data first. These nodes are the nodes which have no outgoing edges. These nodes demand no data but are only demanded for data. If the latter is not the case, these nodes were useless and they would have been eliminated from the demand graph by the demand graph optimiser.

The initial marking for the petri net is formed by putting a token in the output places of these nodes. However, these nodes are not mapped onto a transition in the petri net, since they have to fire first and may never fire again. These nodes have no input places, so, in the petri net, they would always be enabled to fire which in fact may not happen. Therefore, they cannot be mapped onto a transition in the corresponding petri net.

5.2.1 Anticipation of subtrees in the reachability tree
Computing all possible reachable markings from an initial marking results in a very large reachability tree. Therefore, the construction of a complete reachability tree is very time consuming. Since many subtrees in the reachability tree give no new information about the behaviour of the petri net, these subtrees may be deleted from the tree. Better, these subtrees should not be computed when constructing the reachability tree. The construction of subtrees which yield no new information will not be done when these subtrees are anticipated as described in chapter . The field anticipated-transitions of a node indicates which transitions are anticipated at this state of the petri net.
The algorithm used for the anticipation of enabled transitions is given by Algorithm 2.

**Algorithm 2 Anticipation of enabled transitions**

```plaintext
enabled_transitions := sort(enabled_transitions)
construct_reach_tree(first(enabled_transitions))
enabled_transitions := anticipate(first(enabled_transitions))
rest(enabled_transitions))

enabled_transitions := rest(enabled_transitions)
while enabled_transitions do
    construct_reach_tree(first(enabled_transitions))
    enabled_transitions := anticipate(first(enabled_transitions))
    rest(enabled_transitions)
endwhile

function anticipate(prev_transition enabled_transitions)
begin
    for all transitions in enabled_transitions do
        if and (not(conflict_transition(prev_transition))
            not(conflictive_pair_of_transitions(prev_transition))
            or(transition fired after prev_transition
                transition anticipated at prev_transition)
        then
            delete transition from enabled_transitions
            insert transition in anticipated_transitions
        endif
    endfor
    return(enabled_transitions)
end
```

Before the reachability tree is constructed, the conflictive transitions are computed. Thus, the determination whether a transition is in conflict with another transition can be done very easily. When a state of the reachability tree must be processed, first the enabled transitions are computed. These transitions are in such a way sorted that first the conflict free transitions will be fired. As described before, this will result in a smaller reachability tree than firing conflictive transitions first. Transitions, even conflictive transitions, may be anticipated if the first subtree starts with a conflict free transition. Control transitions are dealt with identical to conflictive transitions, since the action of the control transition depends on the value of the control place of such a transition. Therefore, the token flow in the petri net is not always the same. This may also be the case for conflictive transitions, since one enabled transition will fire which disables the other enabled conflictive transitions. So, the marking may be identical for two states of the petri net, but the new marking may be different, even if the same control transition fires.

First a subtree at the current state is constructed. After that, the other enabled transitions which may be anticipated are removed from the list of enabled transitions and are inserted in the list of anticipated transitions at this state. Now the next subtree will be constructed. After that, it is again checked which transitions may be anticipated. This continues, until the list of enabled transitions is empty. As mentioned, the list of enabled transitions can be made smaller by firing an enabled transition or by anticipating the transition.
Also, the concept of multiple firing of transitions described in chapter may reduce the complexity of the reachability tree. Multiple firing does not mean that one transition fires more than once at a time, but that the order of firing several transitions does not matter the behaviour of the petri net. Multiple firing has much to do with anticipation. More than one transition may be fired, if these transitions may be anticipated. The anticipation must be independent of the order of firing. So, all transitions must be conflict free transitions. Since the order of firing these transitions does not matter, an order is chosen and the result is only one node in the reachability tree instead of a tree.

5.2.2 Reduction of operator nets

Another very useful reduction of the complexity of the reachability tree can be made by mapping several nodes of the demand graph onto one transition in the petri net. The reduced petri net has the same properties like liveness and safeness as the original petri net if the petri net which corresponds to this set of nodes has these properties. This condition is valid for the operator nodes in the demand graph, since they are connected in a 'straight' line with a well defined data flow which will never result in deadlocks or unboundedness. Since the nodes which constitutes a loop in a petri net are not operator nodes, these nodes are not considered in the reduction of the petri net. Thus, the operator net is well behaved since loops can not occur in such a net.

A subgraph of operator nodes is mapped onto a minimal set of transitions in the corresponding petri net. All different outputs of a subgraph, i.e. different nodes on the lowest level of the graph, are mapped onto a new node in the graph, while the other nodes in the subgraph are deleted. The new nodes are mapped onto transitions in the petri net. Each new output node of the subgraph performs the action of the subgraph, i.e. the paths from this node to the input nodes of the operator subgraph which it represents.

The result of such a reduction in a demand graph is shown in figure 5.1.

![Figure 5.1. Reduction of an operator network](image-url)

The algorithm for reducing the operators is given in Algorithm 3.
Algorithm 3 Reduction of operators

for all operator nodes in graph for which the next nodes are not operator nodes do
reduce_operator(operator_node)
endfor

procedure reduce_operator(node)
begin
for all previous operator nodes do
node.prev_nodes := node.prev_nodes - {previous operator node}
node.prev_nodes := node.prev_nodes + reduce_operator(previous operator node)
endfor
end

This algorithm is implemented, but with the removal of reduced operator nodes and without reducing the same operator more than once. Therefore, the algorithm for reducing the operators in a demand graph is linear in the number of operator nodes.

Also, some other nodes of the demand graph may be reduced: the nodes which form a link between so-called control nodes and the rest of the demand graph. These nodes are necessary in the demand graph, since, for example a merge node, has only one output for each possible value of the control edge. But, if more than one node demands data of this node, this would not be possible in this case. Therefore, nodes are introduced which are the link between nodes which demand data of a control node and the control node itself. In principle, these link nodes have no other function than passing data from the control node to other nodes. In the petri net which corresponds with the demand graph, there is no need for such link nodes, since the output transitions of the control transition can be the transitions which should demand data of the control transition.

Figure 5.2. Removal of link nodes in the petri net a) demand graph b) corresponding petri net

Such a simple net of the control transition, the corresponding link transitions and the transitions which demand the data is a well behaved petri net. This net may be replaced by a petri net which consists of the control transition and the demanding transitions. So the link transitions are moved into the control node. This reduced petri net has the same behaviour as the petri net in which these link nodes are not removed. See also figure 5.2.
5.2.3 Timing properties of the petri net

Since one should deal with timing information in the verification of systems, the petri net must also be able to deal with timing information. This timing information may be delays for nodes. But the timing information may also be some constraints between transitions, for instance: transition $t_i$ must be fired before $t_j$, or some explicit time information as transition $t_i$ must fire $\tau$ seconds before transition $t_j$.

Therefore, the petri net is in fact a time petri net. Each transition has a minimum and a maximum time in this time petri net. These minimal and maximal times indicate in which time interval the transition may fire after it gets enabled. In chapter, it is indicated how one may find the transitions which are enabled at a moment and the resulting timing information after firing of such a transition.

5.2.4 Further implemented analysis tools

Besides the reachability tree, reductions of this reachability tree and the implementation of time petri nets, some other analysis tools are implemented. These tools are:

- verification of boundedness and safeness
- verification of liveness
- returning of timing constraints under which the reachability tree may behave correctly.

After the reachability tree has been computed, these properties of the reachability tree can be analysed and are discussed in the next paragraphs.

5.2.4.1 Boundedness and safeness

Boundedness may be verified for several bounds, i.e. the maximal number of tokens in a place. When the bound is chosen to be one, safeness of the petri net is verified. The check of boundedness is simply done by checking the boundedness for each marking in the reachability tree.

5.2.4.2 Liveness

Liveness of the petri net may be verified at several levels of liveness, since more than one definition of liveness of a petri net exist. The implemented levels of liveness are:

- Level 0
  A transition in the petri net exists which is never enabled to fire in the reachability tree. Thus, a dead transition exists.

- Level 1
  All transitions of the petri net are at least once enabled to fire in the reachability tree.

- Level 2
  All transitions of the petri net may fire at least $n$ times for every integer $n$.

- Level 3
  All transitions of the petri net may be fired infinitely often in an infinite firing sequence in the reachability tree.

- Level 4
  All transitions of the petri net are potentially fireable from each marking in the reachability tree of the petri net.

Not only these levels of liveness for a petri net are implemented, but also liveness checks for only one transition. Thus, one can also verify which transition is live at which level. A petri net is live at a certain level if all transitions of that net are live at that level. In general, a non-live net is called a dead net. However, when liveness at
level 0 is concerned, this does not make any sense. Therefore, liveness at level 0 is not of concern since it can directly be derived from liveness at level 1. If the petri net is live at level 1, it is not live at level 0 (so it is live). If the petri net is life at level 0, it is dead at level 1 (and it is dead).

5.2.4.3 Timing constraints

From the reachability tree, one can see if unwanted firing sequences may occur. If this is the case, one can give timing constraints to transitions of the petri net, so that these illegal sequences never occur. This may the case if the petri net expects input which will not be available at all the times the net is expecting it. Also, data may be put too early to the environment of a subsystem, so that the previous data gets clobbered. In most cases, the timing constraints will not be absolute timing constraints, but relative constraints. So, one transition has to fire before another transition. But if each transition has an ordinary minimal and a maximal time between it will fire, then the timing constraints become absolute expressions.

One may also examine from the reachability what the minimal and maximal time of a particular firing sequence is. Thus, if this time is not satisfactory, one should alter the timing figures of one or more transitions. It could also be possibly to give constraints which depend on other transitions, so that this particular sequence cannot occur. The computation of the minimal time and the maximal time of a firing sequence can be done from the domain equation of a marking. A domain of a marking at which transition \( t_i \) will fire yields that

\[
\begin{align*}
\tau_i \leq \tau_j \\
a_i \leq \tau_i \leq b_i \\
a_j \leq \tau_j \leq b_j \\
\tau_i - \tau_j \leq c_{jk} \\
\tau_i - \tau_j \leq c_{ji} \\
\tau_j - \tau_i \leq c_{ij}
\end{align*}
\]

From this set of equations, it can be derived that

\[
\begin{align*}
\tau_{i_{\min}} &= \max (a_i, a_j - c_{ji}) \\
\tau_{i_{\max}} &= \min (b_i, b_j, c_{ij} + b_j)
\end{align*}
\]

with \( \tau_{i_{\min}} \) the minimal delay of transition \( t_i \) to fire after it gets enabled and with \( \tau_{i_{\max}} \) the maximal delay.

Of course, \( \tau_{i_{\min}} \) has to be less than \( b_i \) and this is also valid in the above equations, since \( a_j - c_{ji} \leq a_j + b_i - a_j = b_i \). This last equation is valid since the domain is canonic, i.e. \( c_{jk} + a_j \geq a_j - b_k \). Also, \( \tau_{i_{\max}} \) has to be larger than \( a_i \). This is true, since \( b_j \geq a_i \) (since transition \( t_i \) fires before transition \( t_j \)) and \( c_{ij} + b_j \geq a_i - b_j + b_j = a_i \) (since the domain is canonic).

Also, suggestions can be made for timing constraints of transitions which lead to non-safe markings. This is done by searching for the non-safe markings. At each non-safe marking, it is checked if another marking sequence exists with safe markings beginning from the parent of this marking. Of course, such a tree may not begin with a transition which is in conflict with the transition leading to the non-safe marking, since this results in a different behaviour of the petri net.
Also, the returning of timing constraints for the availability of data is implemented. This is done by searching the markings in the reachability tree which result in a correct behaviour. The two possibilities are compared with each other, so that a timing constraint can be extracted. This timing constraint guarantees that the wrong behaviour will not show up in this case.

5.3 Implementation of markings

The state of a normal petri net is the same as the marking of the petri net. In a time petri net, the state is also formed by the timing constraints of the petri net in that state. A marking is an assignment of tokens to the places of the petri net. Such a marking can easily be implemented in a vector. Each element of the vector corresponds to a place of the petri net. The firing of an enabled transition results in a new marking. This new marking can be generated from the old marking by adding the input vector and the output vector of the transition to the old marking, as indicated in the next equation:

\[ \mu' = \mu + T^- + T^+ \]

In this equation, \( \mu \) is the marking before firing of the transition, \( \mu' \) after firing, \( T^- \) is the input vector the transition and \( T^+ \) the output vector. Therefore, the computation of markings after firing a transition can easily be done by adding vectors to the vector of the marking before firing.

Such a vector representation has been made for the markings of a reachability tree. However in most cases, only a few places of the petri net have tokens. Thus, many elements of the marking vector are zero. Also, transitions have only a few input places and a few output places and therefore the input and output vectors of a transition contain a lot of zeros. These two facts and the fact that a large number of places exist in a petri net, imply that adding two vectors results in a lot of redundant computations, namely to add zeros.

For this reason, a new representation for a vector has been made, so that no time is lost for adding zero tokens in a place. Now a vector is a list which contains only information about the places which have tokens. Each element of this list is a pair. A pair consists of the place and the amount of tokens in that place. Each place in the list contains at least one token. For adding two vectors, only the places which contain tokens are considered. This results in less computations than when all places are considered. In the examples, it is shown that the time needed for constructing the reachability tree is really reduced, especially when the petri net has a lot of places.

5.4 Complexity of the construction of the reachability tree

The reachability tree of a petri net consists of all possible states of the petri net. Therefore, the time and space requirements for the construction of the reachability tree is exponential. Anticipation of the conflict free transitions of the petri net may reduce the complexity of the reachability tree. However, in worst case no reductions can be done and the complexity is still exponential. In the best case, the petri net has only one behaviour and every possible path starts at the initial marking and ends in only one final state. Therefore, all paths but one may be anticipated. So, in best case, the space and time requirements for the construction of the reachability tree are linear in the number of transitions of the petri net.

The petri net is constructed from a demand graph with each node of the demand graph mapped onto a transition in the petri net. Therefore, the complexity is also linear in the number of nodes of the demand graph. Operator nodes may be removed out of the demand graph since they do not give new information about the behaviour of the corresponding petri net. Also, the link-in nodes of the demand graph do not have to be modelled by transitions in the petri net.
The communication of the system is also modelled in the petri net of the system. The number of transitions in this petri net is equal to the number of connections. Therefore, the complexity of the reachability tree is in best case linear in the number of nodes of the reduced demand graph and the number of and connections.
6. Results

Functional descriptions have been made for several examples to test the method of verification presented in this report. This means that the functional descriptions have been converted to demand graphs and to petri nets. Also protocols have been defined and converted to petri nets. These petri nets are analysed. The results of the analysis are checked on their correct behaviour. Correct behaviour means that all possible behaviours of the net correspond to the specified functional description. Also, it is checked that the net should be free of deadlocks, be safe etc.

For several examples, this verification traject has been gone through and are enlisted below:

- Tseng example
- Simple communication example without any protocol
- Two phase handshaking communication protocol

These examples will now be discussed in the next paragraphs. Extensive results can be found in the appendices.

6.1 Tseng example

This example can be found in [Tsen86] and was meant to illustrate the hardware synthesis phase of a silicon compiler. The example consists only of arithmetical and logical operations. Of course, it inputs some data and outputs the computed results. Therefore, this example will not show anything about the verification of communication. It will only illustrate the reduction in the state space when two reduction techniques are applied to the example. The used reductions are the anticipation of conflict free transitions and the reduction of operators. The full results of this example can be found in Appendix 1.

Since the petri net of this example does not have any conflictive transitions, only one subtree has to be constructed at any node in the reachability tree. Other subtrees do not contribute new information about the behaviour of the petri net and may therefore be anticipated. This anticipation results in a much smaller reachability tree. As one can see in the original reachability tree, all paths from the root of the reachability tree lead to one terminal marking. Therefore, the anticipated reachability tree consists of only one path from the root to this terminal marking.

6.2 Simple communication example

This example shows how communication is dealt with in the analysis. The example consists of two subsystems which communicate with each other and no protocol has been defined. The verification method must give timing constraints if the circuit does not behave correctly in all possible cases. In this particular example, this means that no input may be expected if no data has been sent. Also, it may happen that a first message is clobbered by a second message before the first message has been read. As shown in the results, these conditions are detected by the verification. Therefore, timing constraints are given under which the circuit will behave properly. The results show also that the circuit will indeed not show the detected faulty behaviour when the returned timing constraints are applied to the system. The timing constraints assure that the system will behave properly and therefore, the hardware synthesis phase of the silicon compiler should take care of the implementation of these timing constraints.

The reachability tree of the system with timing constraints is much larger than the reachability tree of the system without timing constraints. One should expect a smaller reachability tree when restrictions are applied. However, this is not the case. It probably would be if anticipation of conflict free transitions is not done in both cases.
When several timing constraints are applied to the system, the anticipation cannot reduce as much as when no constraints are applied. This is due to the fact that domains, i.e. the systems of inequalities which describe the timing intervals of enabled transitions, are not equal for all transitions. Anticipation may be done if the domains are equal. If the domains are not equal, it is not sure if one may anticipate. Therefore, anticipation of conflict free transitions is only done when domains are equal.

Also, states are only equal if their markings and their domains are equal. This causes a larger reachability tree also.

As one can see, the reachability tree contains paths to an earlier processed state. These paths should not be constructed in the reachability tree. Therefore, some rules have to exist that these paths can also be anticipated. Probably, these rules are the same as the used ones, but which are less strict in the equality of domains. The implemented anticipation does only anticipate conflict free transitions if the domains are equal for these transitions. However, it is possible that this constraint is too strict and this rule should therefore be loosened.

6.3 Two phase handshaking protocol

This examples shows the behaviour of a two phase handshaking protocol. This protocol is defined in such a way that first a request must be sent from the system which sends a message to the receiving system. The receiving system must first acknowledge the sending system. The data may be sent only after the acknowledge has been received. In this case, the two communicating systems have made a connection, so that the message will arrive correctly at the receiving station. Such a protocol may be necessary, since the data communication channel may be used for other purposes than the communication of data between the two systems. Therefore, this channel must be free and the protocol should guarantee this.

The protocol is a handshaking, since both communicating systems are involved. One system will give a request to send, while the other system must acknowledge this request. Therefore, all the involved systems must grant the permission to transfer data.

The described protocol is a two phase protocol instead of a four phase protocol, since the request and acknowledge signals have only one signal value change during the whole cycle of the protocol. Therefore, only two phases can be distinguished in a communication cycle as shown in figure 6.1. In a four phase handshaking protocol, the request and acknowledge signals return in the same state as before the communication protocol mechanism started. Four phases can therefore be distinguished in such a protocol as indicated in figure 6.2.

![Figure 6.1. Two communication cycles in a two phase handshaking protocol](image)

In a two phase handshaking protocol, each two communication cycles has a different set of signal values for the request and acknowledge signals, since only one value change marks a communication. In the verification of the two phase communication protocol, only one cycle is verified. The verification of the other communication cycle will not reveal new information, since only the signal values are changed. Extensive
Figure 6.2. One communication cycle in a four phase handshaking protocol results of the verification of this communication protocol can be found in Appendix 3.

The analysis of this handshaking protocol shows that it behaves correctly. The data sending system will only start the data transfer when it has been acknowledged by the receiving system. Therefore, no faults can occur in the communication. No timing constraints need to be given for the correct behaviour of the protocol since the protocol functions properly independent of the timing of the separate elements, provided that the acknowledge is read after the request has been sent.
7. Conclusions

The silicon compiler, which is in development in this department, is meant to generate a layout from a functional description of the behaviour of a system. It will generate synchronous circuits as the hardware implementation of the functional description. For large systems, the systems will be divided in subsystems which have to communicate with each other in an asynchronous way. The functional description of the system has to be extended with just one new basic element to deal with asynchronous features also, instead of synchronous systems only. However, the internal data structure of the silicon compiler does not need any extension, since asynchronous communication can be modelled with the set of already existing elements.

Protocols have to be defined for the asynchronous communication between interacting systems. They are needed to assure a correct behaviour of the whole system. In principle, not specifying a protocol can also yield a correct behaviour of the system.

In this report, a verification method is presented. This method verifies whether a protocol behaves correctly according to its environment. The environment of a protocol consists of the interacting systems for which the protocol is defined. The two major properties of a proper communication protocol are the facts that no deadlocks may occur and that the protocol is safe. A safe protocol means that messages do not get clobbered. The chosen verification method is the method which uses petri net analysis techniques. Therefore, the system is seen as a petri net, which is analysed. The analysis of the behaviour of a petri net is done with help of the reachability tree of such a petri net.

Petri nets can deal easily with properties that are important in communication. Therefore, petri nets are chosen as the method for verification. Also, petri nets are closely related to demand graph and the demand graph is the main data structure in the higher levels of our silicon compiler. Since protocols can be defined in the same way as the functional description of the system, they can be converted into a demand graph, too. In that case, nothing special needs to be done for the combined verification of the protocols and the system itself, since everything is combined in a single model.

The verification method does not just only answer the question whether the system behaves according some properties, but may also give suggestions for constraints under which the system behaves correctly. When these constraints are applied to the system, it may be that the system behaves just as it should be. Therefore, it could be possible to generate constraints under which the communication without any specified protocol functions properly. These timing constraints indicate in most cases which transitions must be fired in what order. Thus, these constraints extract some non-determinism out of the original system. The hardware synthesis phase of the silicon compiler must take care of the constraints which are given by the verification to assure that the generated circuit functions properly, according to the specified functional description.

The implemented verification method for a petri net gives good results, but it costs quite a lot of memory and time. This is due to the fact that almost all possible states of the petri net are computed and this results in a giant state space. Therefore, some state space reductions are made and implemented. These reductions result in a reachability tree which is much smaller and more tractable. Therefore, analysis can be done much faster on such a reduced state space.

However, in the results it is shown that the reachability tree may be still large, especially when timing constraints are applied to the system. Therefore, new reductions have to be found and/or the implemented reductions have to be extended, so that the reachability tree stays small. One should investigate first, if one may anticipate conflict free transitions even if domains of a state are not equal. Currently, anticipation is only done if domains are equal, but it may be possible that conflict free transitions may be
anticipated when domains are not equal. If one can state rules for the domains, this will reduce the space and time requirements to construct a reachability tree which contains all necessary information. These rules must guarantee that anticipation does not lead to loss of information. It is very probable that such rules exist, since the results show that the reachability tree contains many paths to the same marking. All these paths give no new information about the behaviour of the system. Therefore, one should look for more reductions to construct a reachability tree as small as possible.

One should distinguish a data network and a control network in the system. Only the control network of the system is important for the verification. The data network just shows how data is computed but it does not influence the behaviour of the system, since this is caused by the control network. Of course, the data path may contain data dependent decisions. The control network indicates only which data computations are done by which subsystem and controls the interaction between the several data computing parts in the system. Also, the data part of the system will in general be verified before the functional description is really applied to the silicon compiler. For instance, this verification can be done by executing the functional description when it is written in a real programming language. The control network of the system just controls what task on the data should be performed. This control network must therefore be carefully checked. If the data network is not considered in the verification as described in this report, this will also reduce the complexity of the analysis. The data network may in that case be modelled as one transition, and the verification results may give constraints to this data network. These constraints must be taken into account in the realisation of the data network into a physical circuit.

However, the analysis of a petri net may still consume rather a great amount of time. Therefore, one should search for other useful reductions of the petri net and of the computations which are needed for the analysis of the petri net.

The verification method is implemented to analyse the whole petri net at once. However, a more interactive approach can be managed with only a few adaptations to the analysis tools. This means that the verification may be done 'on the fly'. At this moment, all necessary possible states are computed and this includes all the states which are not safe or bounded. However, one can easily implement a halt in the computation of the possible states whenever such a state is discovered. One could generate the constraints for this state and start all over instead of computing the other possible states. Also, one may easily implement that not all the reachable states from a non-safe marking are computed. All these efforts will reduce the time and space complexity of the analysis. In the best case, the complexity of the construction of the reachability tree of a petri net will be linear in the number of nodes of the reduced demand graph.

An important feature of the verification method with help of petri net analysis methods is the fact that it may be applied at more stages of the automatic synthesis than the one described in this report. Petri nets do not only suit very well for demand graphs but also for other data structures. For example, a finite state machine can be easily modelled as a petri net. Since a petri net is able to model non-determinism, a non-deterministic finite state machine can also be dealt with. A result of the hardware phase of the silicon compiler is a controller. This controller is a finite state machine and it controls the mapping of operations onto modules. The controller indicates when what operation on which module will be performed and where it can find its input data and where to put its outputs data. Also, it controls the function of the module if the module can perform more than one operation. This finite state machine can easily be converted into a normal petri net and this net can be analysed with the same methods as described here. However, the same limitations of the analysis method is valid for such finite state machines as for the petri nets described in this report.
Another great advantage of analysis methods which are based on petri nets, is the fact that it can be used for a lot of applications, since the concept of petri nets is a very strong modelling tool. Really a large number of systems can be modelled as petri nets, and therefore petri net analysis methods can be applied to these systems, especially if properties like liveness and boundedness have to be verified.
References

[Aspv79] Aspvall B. and Y. Shiloach
"A Polynomial Time Algorithm for Solving Systems of Linear Inequalities with Two Variables per Inequality"

[Bert83] Berthomieu B. and M. Menasche
"An Enumerative Approach for Analyzing Time Petri Nets"

[Eber87] Ebergen J.
"Translating Programs into Delay-Insensitive Circuits"

"Synthesis of Structure and Logic under VLSI conditions"

"Optimisation Steps in Silicon Compilation"

[List85a] Lister P.F., C. Eng and A.M. Alhelwani
"Design Methodology for Self-Timed VLSI Systems"

[List85b] Lister P.F. and A.M. Alhelwani
"Data-flow Based Design of Self-Timed Systems" IEE Colloquium on VLSI Design Methodologies, IEE Digest No. 41, April 1985, pp. 4/1-4/4.

[Mena85] Menasche M.
"PAREDE: An Automated Tool For The Analysis of Time(d) Petri Nets"

[Merl74] Merlin P.
"A Study of the Recoverability of Computing Systems"

[Mura77] Murata T.
"Circuit Theoretic Analysis and Synthesis of Marked Graphs"

[Noe73] Noe J.D. and G.J. Nutt
"Macro E-Nets for Representation of Parallel Systems"

[Pech76] Pechoucek M.
"Anomalous Response Times of Input Synchronizers"

[Pete81] Peterson J.L.
"Petri Net Theory and the Modelling of Systems"
[Petr62] Petri C.A.
"Kommunikation mit Automaten"

[Ramc73] Ramchandani C.
"Analysis of Asynchronous Concurrent Systems by Petri Nets"

[Rem87] Rem M.
"Het Vertalen van Parallelle Programma’s in Geïntegreerde Schakelingen"
Proceedings of the Symposium on Silicon Compilation, organised by IEEE
section Benelux, Nederlands Elektonica- en Radiogenootschap, and Konink-
lijk Instituut voor Ingenieurs, September 16 1987, Delft. (not yet published)

[Seit80] Seitz C.L.
"System Timing"
Chapter 7 of "Introduction to VLSI systems", C. Mead and L. Conway,
Addison Wesley, New York 1980

[Sifa79] Sifakis J.
"Performance Evaluation of Systems Using Nets"
Proceedings of the Advanced Course on General Theory of Processes and

[Stok86] Stok L.
"From IDEA to IC, the higher levels of a silicon compiler"

[Tsen86] Tseng C.J. and D.P. Siewiorek
"Automated Synthesis of Data Paths in Digital Systems"
Appendix 1: Tseng Example

The Tseng example [Tsen86] has the following functional description. This description is written in the format of the abstract syntax tree [Stok86].

(program tseng
  () (v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15)
  (: (get p1 v10 v6)
      (get p1 v4 v2 v1)
      (:= v3 (+ v1 v2))
      (:= v5 (- v3 v4))
      (:= v7 (* v3 v6))
      (:= v8 (+ v3 v5))
      (:= v9 (+ v1 v7))
      (:= v11 (/ v10 v5))
      (:= v12 100)
      (:= v13 v3)
      (:= v12 v1)
      (:= v14 (and v11 v8))
      (:= v15 (or v12 v9))
      (:= v1 v14)
      (:= v2 v15)
      (put p1 v1 v2))
  )
)
This example has the demand graph which is shown in the next figure.

The nodes and edges of the demand graph are one to one mapped onto transitions and places of the corresponding petri net. Some reachability trees are made for this petri net to show the reduction techniques of the petri net analysis methods. In all these examples, the primary input nodes have been initially fired. This results also in a reduced reachability tree.
Reachability tree without any reduction at all:
When the operators are reduced in the original demand graph, the following demand graph is left:

![Diagram of demand graph with operators and edges labeled.

Reachability tree for this reduced example:

![Diagram of reachability tree with nodes and edges labeled.]}
The reachability tree of the original petri net but with anticipation of conflict free transitions:
The reachability tree of the reduced petri net with anticipation of the conflict free transitions:
Appendix 2: Simple communication example without any protocol

A communication example is presented. The communication in this example is specified by no protocol at all. Therefore, the verification method should detect the possible faulty behaviour of the example and it should give constraints under which it will behave correctly.

The example consists of two subsystems which communicate with eachother. The syntax tree of the example is:

```
(program simple-comm1
) (a b h1 h2)
; (get port-1 a)
(get port-2 b)
(= h1 (+ a b))
(put port-3 h1)
(= h2 (+ a h1))
(put port-3 h2)
)
```

```
(program simple-comm2
) (c d e h3)
; (get port-4 c)
(get port-5 d)
(get port-6 e)
(= h3 (+ c d))
(= h4 (+ e h3))
(put port-7 h4)
)
```
The demand graph of this set of systems is:

The dashed lines in the demand graph mean that the input node expects data from the output nodes indicated by the line. These lines represent the connection between the ports of the input node and the output nodes.
Since the reachability tree without anticipation is too large, the reachability tree with anticipation of conflict free transitions has been computed. This resulted in the following reachability tree.

This reachability tree indicates that the input node for variable $e$ could be implemented before it should be. Therefore the following timing constraint is suggested:
take care that transition D-12 is fired after transitions D-2, D-3, D-4 and D-5

The system will not show this faulty behaviour if this constraint is applied to the system.

This example may show two other faulty properties which are also indicated by the reachability tree analysis. These incorrect behaviours are: non-safeness of data in ports port-3 and port-6. This is due to the fact that subsystem simple-comm1 may output twice data before subsystem simple-comm2 inputs the data it needs. The suggested timing constraint is:


take care that transition D-7 is fired after transitions C-0 and D-12.

The following timing constraints are applied to the system:

D-2 firing interval <0,1>
D-3 firing interval <0,1>
D-4 firing interval <0,1>
D-5 firing interval <0,1>
D-7 firing interval <12,\omega>
D-12 firing interval <10,11>

The other transitions have firing intervals <0,\omega>
In this case, the (anticipated) reachability tree is:

No wrong behaviour can be detected in this reachability tree. Therefore, when the timing constraints are applied to the system, it behaves correctly.
Appendix 3: A two phase handshaking protocol

A two phase handshaking communication protocol is presented here. The system that wants to send a message must first send a request to the receiving system. The data transfer may be started when the request has been acknowledged.

In the following, only one communication cycle is analysed. Also, the data which has to be transferred is not made explicit.

The syntax trees for the two systems are:

\begin{verbatim}
(program send () (request acknowledge data)
  (; (:= request 1)
   (put port-1 request)
   (on (get port-2 acknowledge)
     (put port-3 data)
   )
  )
)

(program receive () (request acknowledge)
  (on (get port-4 request)
    (; (:= acknowledge 1)
     (put port-5 acknowledge)
    )
  )
)
\end{verbatim}

The demand graphs of these two systems are\(^1\):

\begin{center}
\includegraphics[width=0.7\textwidth]{demand_graph.png}
\end{center}

\begin{verbatim}
req  port-2  ack  port-3  data
\end{verbatim}

\footnote{1. All entry and exit nodes have control edges to the 'not' operator.}
Two connections are in this example. One connection \((C-0)\) is between \(port-5\) and \(port-2\). This connection transfers the \(acknowledge\) signal. The other connection \((C-1)\) connects \(port-1\) and \(port-4\) and transfers the \(request\) signal.
When the corresponding petri net is analysed, the following reachability tree is obtained.
Three phases can be discovered in this reachability tree:

I  The sending system outputs the request
II The receiving system computes and outputs the acknowledge
III The sending system receives the acknowledge and outputs the data

This protocol does not have incorrect behaviour. Therefore, no timing constraints are necessary to assure a correct behaviour.