MASTER

An implementation of reactive process networks

Alberga, D.

Award date:
2008

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Design Technology For Electronic Systems (ICS/ES)
ICS-ES 893

Master's Thesis

AN IMPLEMENTATION OF REACTIVE PROCESS NETWORKS.

Daan Alberga

Supervisor: prof.dr. H. Corporaal
Coach: dr.ir. M.C.W. Geilen
Date: February 2008

Contents

Summary

Kahn Process Networks (KPNs) are a good way to model streaming data applications, like multimedia applications that are quite popular nowadays. A major drawback of KPNs however, is that the function that KPN processes execute remains constant, although it can have parameters. Real reconfiguration, i.e. a real change of the function that is executed, is not possible while the execution of the network continues, as is a synchronized way of reconfiguring several processes at once. The RPN model [5] gives a solution, but remains very theoretical and doesn't solve problems like how reconfiguration should be organized without stopping and emptying the whole network. The proposed model solves this problem by using a different perspective of time: tokens are the time units, instead of seconds or milliseconds. Several ways of reconfiguration, both synchronized and non-synchronized, are investigated. The network can now be reconfigured while it is in operation. This theoretical model is described in chapter 3.

The existing C++ toolbox YAPI has been modified to be able to implement these new networks and to simulate them. How this works is explained in chapter 4. Chapter 5 gives a manual to implement new process networks.

Chapter 1

Introduction

1.1 KPNs and RPNs

In 1974, Gilles Kahn introduced the Kahn Process Network (KPN)[6], a computing model, consisting of processes, connected by infinitely large fifo-buffers, that function as communication channels. Every process represents some function, fully unsynchronized and independent of other processes. Communication is only possible through tokens of a certain type in these fifo-buffers. Tokens are information units, that can contain any type and amount of information, e.g. a boolean, an integer, a sound sample or a complete movie. Every channel can only have one type of token.

This model has been popular in computing theory, but hardly was interesting in practice, because it is primarily interesting for multi-processor architectures, which never were really popular. Nowadays however, the interest has grown, because the limits of one-processor architectures are being pushed further and further, making them increasingly expensive, while several slower processors, made with cheaper technology, might together be as good as, or even better than one fast processor. Especially multimedia systems tend to be easily modeled in KPNs, because they usually perform several consecutive operations on the same information.

One major drawback of the KPN model is that it is static. Reconfiguration isn't supported. This is unacceptable for most multimedia applications, because users should be able to adjust the configuration while executing the algorithm, e.g. adjusting the volume or the brightness of a movie. Therefore lately, this has been a research subject. One of the developments have been the paper [5] by Marc Geilen and Twan Basten, introducing the Reactive Process Network (RPN). In this paper, processes can have several states, each executing a different function. These processes switch between those states.

1.2 Assignment

The purpose of the master's assignment that this report is about, was firstly to explore the RPN model and to specify it further, to enhance it, as well as to solve the problems that arise when implementing it. Secondly, the model had to be implemented into YAPI, a C++ tool box that was designed to implement KPNs.

1.3 Report

This report explains the solution that has been found to the problem of the reconfiguration of KPNs, as well as the implementation. The context and the related work that have preceded this report can be found in chapter 2. The theoretical solution is described in chapter 3. Chapter 4 describes the implementation of this model. This chapter also describes which simplifications had to be applied in order to implement the model. Chapter 5 describes how the implementation can be used to build RPN applications. The final chapter gives conclusions and recommendations. Used literature and the program texts can be found in the appendices.

Chapter 2

Related work

In the area of flexible processing networks, much work has already been done. This can be divided into several categories:

1. Works about reactiveness in KPN and SDF.
2. Works about how resources in a computer system should be distributed and redistributed.

For this report, it is mainly the first category that is interesting. Many models in this category, however, suffer from several drawbacks:

1. Choices are made in a non-deterministic way.
2. Large parts of the network are halted to switch to another state instead of continuing working; the network is 'flushed'.
3. The network cannot react on events that happen at unpredictable moments.
4. The input and output rate of processes (tokens per loop) imposes more rigid constraints.
5. The network doesn't provide synchronized reconfiguration

Ad 1: [1] has an interesting view about timing. However, if several tokens arrive at a process at the same time, the choice which event receives priority is made in a non-deterministic way. This is the opposite of what was intended in this report: to keep the determinism of KPN present, except for the arrival of reconfiguration commands.

Ad 2: flushing is proposed in [5], although the details are not filled in in this paper. The model in this report gives a solution for this, so the network can always continue working whenever there is enough data available at a process's input channels.

Ad 3: an example of this phenomenon is SADF [12]. This model is a flexible dataflow system, but it is derived from SDF. SADF cannot react on unpredictable events. It is possible to model reactive behavior in SADF, but only if

every computation would, apart from the input data, receive some 'reconfiguration token'. This is a crucial difference: the model described in this report only needs reconfiguration commands at moments at which it needs to carry out a reconfiguration.

Ad 4: StreamIt [13] has the drawback that all flow rates in the streams must be static. The model proposed in this report partly solves that: the rates can be dynamic, as long as they are predictable.

Ad 5: Parameterized KPN [9] has this drawback; there is no way of making sure that the processes change at exactly the correct point in time. Another drawback is that the system has to halt before things are changed.

[8] works with a hierarchical model. This is comparable to the subnetwork model that is used in this report. It introduces the quiescent state. A drawback is that it cannot handle stateful reconfiguration.

Much work has been done about resource allocation, like the work of Vincent Nollet [10] and Eclipse [11]. This work is applicable to the system that is described in this report when it is implemented on a multiprocessor platform.

A lot of work had been done on expressing reactiveness in programming languages, like [2] and [3]. Articles like these give a clear view about how networks can be expressed in a programming language. This will be useful if YAPI would be revised at some point in time, as would be [4], an article that focuses on the technical details of implementing streaming applications on multi-processor architectures.

So the work in this reports adds an important new view on the design of reconfigurable dataflow systems. It keeps the system deterministic in every sense, except for the reconfigurations, because reconfiguration events can never be predicted. It provides more freedom in the input and output rates of tokens in channels and it can continue working while the network is computing.

Chapter 3

Theory

3.1 Introduction

The theoretical part of this report starts with a very brief description of a KPN and the question what reconfiguration is and how this is done, stating the central problem. From this point, the report gradually works towards a new perception of process networks, where time is no longer treated in a traditional way, measured in seconds, but in a new way: measured in tokens. The main focus is on how information travels through the network and how this can be expressed. This is treated in section 3.3. The next section is about the reconfiguration of the networks. The chapter ends with some more practical considerations about what is needed for each way of reconfiguration.

This theoretical chapter is an independent section, which means that the implementation isn't mentioned here. The next chapter will explain how this model is implemented and which adjustments have been made to use it in practice.

3.2 Reconfiguration and KPNs

Consider an abstract model of a dvd reading system, figure 3.1. Now at a certain moment, a user wants to resize the window size of the video screen. To be sure that the subtitles are resized at exactly the same moment as the video, both processes that are marked with an arrow should change something at the same time. The trouble is that all processes work independently. That means that the subtitle process might be a few frames (or more) further than the video scaling process. The final goal is to have the processes reconfigured in such a way, that the user sees the size of the video and the subtitle change at the same moment. This way of changing is called 'synchronized' changing.

The video system can be further abstracted to a general network consisting of processes and channels, like a KPN. Every process represents some function $(o_1, \ldots, o_N) = f(i_1, \ldots, i_M)$ for some number of inputs M and some number of outputs N. The processes only communicate through channels that are infinitely large fifo-buffers. Each channel can only contain tokens of one size and type.

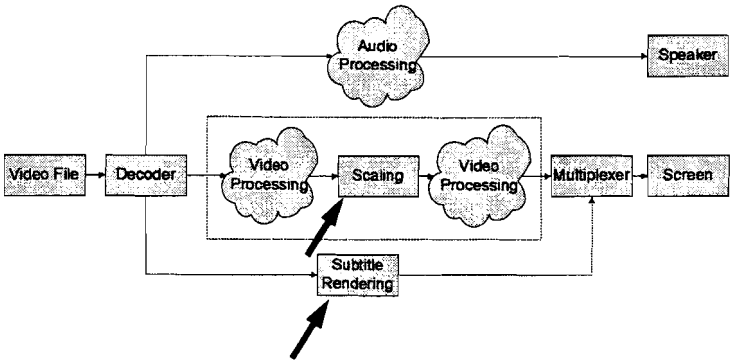

Figure 3.1: A dvd system

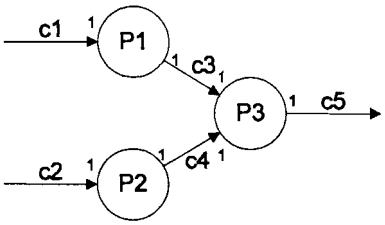

Figure 3.2: The KPN from example 1

Now several processes have to change their functionality in a synchronized way, like in the dvd example. When only one process has to be adjusted, it can basically change at any time. When two or more processes have to be reconfigured in a synchronized way, then there has to be some kind of synchronization between these processes. This isn't a trivial problem, as nothing is known about how many tokens each process will have produced at any time.

Example 1 *Consider a KPN with three processes: processes* $P_1 : c_3 = f_1(c_1)$ *and* $P_2 : c_4 = f_2(c_2)$ *both receive output from the outside world and deliver their output to process* $P_3 : c_5 = f_3(c_3, c_4)$*, see figure 3.2. For this example, it is known that* P_1, P_2 *both produce one token for each consumed token and that* P_3 *alternately picks one token from* c_3 *and one from* c_4 *and then produces a token. Now at a certain moment in time, we want* P_1 *to start executing function* $g_1(c_1)$ *instead of* $f_1(c_1)$ *and* P_2 *to start executing function* $g_2(c_2)$ *instead of* $f_2(c_2)$*. This moment has to be chosen in such a way, that* P_3 *consumes the first token from* c_3 *that has been processed according to* $g_1(c_1)$ *(and not according to* $f_1(c_1)$*) immediately after* P_3 *has consumed the last token from* c_4 *that has been processed according to* $f_2(c_2)$*, because otherwise, the output will see several changes one after another. In other words: the tokens that have been processed 'the new way' have to arrive simultaneously at* P_3*.*

Question: We choose the moment t for P_1 to change its function to be $t = \tau$. At which moment does P_2 have to change its functionality? *Answer:* We don't know. Nothing is specified about at which moment P_1 and P_2 process the tokens

that arrive simultaneously at P_3. There is no guarantee whatsoever that it will be at the same moment; at $t = \tau$, P_2 could as easily be 1000 tokens ahead of P_1 as behind. For this reason, time in the traditional sense of the word isn't very useful to determine the moment for more than one process to change. Therefore, we need a new concept of time.

3.3 A Different Concept of Time

As has been illustrated above, time in the traditional sense of the word isn't a good means to express the moment for processes to change, because for KPNs, the time in the outside world doesn't say anything about how many tokens a certain process has produced.

3.3.1 Time Spaces

Time is just a way to express the order of events, whatever they may be, or to express causality. In the normal world, time is also often used to express some duration, which is a logical thing to do, because human beings actually live and experience each second passing by in some way. Channels don't. Channels can only experience two things during their lives: tokens being written and tokens being read. Therefore, the time for a channel is measured in tokens being written. Time is measured from the start at $t = 0$. The first token arrives at time $t = 1$. These "moments" are used like time, but could also be seen as sequence numbers, which is what they actually are. All tokens are part of the "history" of a channel:

Definition 3.3.1 *The history h_c of a channel c is the set of tokens that have thus far been written in a channel.*

For every moment that has passed in channel c since the start of the network, i.e. for every token that has been in channel c, there is one element in h_c.

Definition 3.3.2 *For a channel c, the time t_c that has passed since the beginning at $t_c = 0$, which is also called the* age *of channel c, is equal to the number of tokens that have been written into the channel:*

$$t_c = |h_c|.$$

Tokens are numbered by the moment that they arrived in the channel, so the first token arrives at time $t = 1$, the second at $t = 2$, the third at $t = 3$ et cetera, not matter how long it takes to produce those tokens. This arrival moment is called a "time stamp":

Definition 3.3.3 *The time stamp t_a of some token a is equal to the moment that it arrived in the channel that it is in. (The function $t(a)$ gives the time stamp of token a.)*

The first token in a channel is written on $t = 1$ and, consequently, has time stamp 1. The tokens in a channel form a strictly ordered set. Token a_i is called younger than token a_j (written as $a_i > a_j$) if $t(a_i) > t(a_j)$ (because a_i was born later than a_j) and elder (written as $a_i < a_j$) if $t(a_i) < t(a_j)$. (NB This has nothing to do with the values of the tokens.) If tokens have the same age, then they must be the same token.

Because every channel defines its own time, we say that every channel is in its own *time space.* These time spaces can be related to each other by processes.

A network can be expressed as a directed graph, where the processes are the nodes and the channels the edges. The channels in this graph (and consequently in the original network) form a preordered set. $c_i > c_j$ if edge c_j can be reached from edge c_i. If edge c_j cannot be reached from edge c_i, then no relation can be given. If $c_i > c_j > c_i$ then c_i and c_j are part of a cycle. Cycles are a more complicated matter; they are treated in paragraph 3.3.5.

3.3.2 Stateless and Stateful Processes

The simplest processes just take some fixed number of tokens from the incoming channels, process them, and output the result as some fixed number of tokens into the outgoing channels, like SDFs do. Having done that, they start over again by again taking some number of tokens from the incoming channels et cetera, without remembering any information from the previous input tokens. This class of processes is called the class of *stateless processes.*

Definition 3.3.4 *A stateless process is a process that handles a certain amount of input tokens (which could also be one) at the same time and then "forgets" the information to proceed to the next amount. So a process is stateless if it executes a function f for which holds:*

$$f(\beta \cdot \beta') = f(\beta) \cdot f(\beta'),$$

where β is a certain sequence in the history of an input channel of f and β' a sequence that follows β. After having computed $f(\beta)$, the process forgets everything - this moment is called a quiescent state*[8]. The process then proceeds to $f(\beta')$.*

There is also another kind of process, which doesn't have the properties of a stateless process:

Definition 3.3.5 *A stateful process is a process that isn't stateless.*

A stateless process can easily be reconfigured at a quiescent state. Reconfiguration means that a certain process starts executing another function than it has before. Stateful processes don't have quiescent states, so they can't easily be reconfigured.

In the dvd-example, the subtitle renderer will probably be a stateless process, because it only needs to read the text of a subtitle to produce an image of this subtitle. It doesn't need the text of the other subtitles as well to do that. A

video processor can be a stateful process, because it might need some image to calculate the next image, so it needs several images to compute the next one. However, this depends on the implementation of the video coding system. Most coding systems will after a certain number of frames return to a state in which they forget all previous data and start again.

3.3.3 Information, Influence, Impact and Sources

Every token that doesn't come from an input, has been produced by a process. Processes are not required to have incoming channels, but they usually do, and if so, at least part of the incoming tokens can be used to produce at least part of the outgoing tokens. When one or more incoming tokens are necessary to produce an outgoing token, then the outgoing token contains *information* from the incoming tokens. This means that the incoming tokens *influence* the outgoing token (which will usually mean that the contents of that token will have some influence on the contents of the token that is being computed):

Definition 3.3.6 *Some token a influences some other token b, if the information from token a is used to create token b.*

For example, if a process has one input channel and one output channel and repeatedly reads two tokens, computes their sum and outputs them, then those two tokens that are summed *influence* the outgoing token that is their sum.

Back to the DVD example: many video coding systems make use of so called I frames and P frames. The I frames are complete pictures, but the P frames only give the difference between a previous I frame and the current picture. When an I frame is decoded, the resulting image will only be influenced by that I frame, but when a P frame is decoded, the resulting image will at least contain information of the decoded P frame and of the last I frame. In some formats, the resulting image will also contain information of all images between the last I frame and the current P frame.

A certain moment τ in a certain time space can be transformed to its corresponding moment in another time space. In the same way, a certain token a in a certain channel can be transformed to its corresponding token in another channel. In forward or backward transformations, tokens are considered 'corresponding' if one influences the other.

Definition 3.3.7 *The* forward *transformation* $a|_j^i$, $c_i > c_j$ *of token a in channel* c_i *returns the first token in channel* c_j *that is influenced by a. The* backward *transformation* $b|_i^j$, $c_i > c_j$ *of token b in channel* c_j *returns the oldest token d in channel* c_i *for which* $d|_j^i = b$. *If that token doesn't exist, it returns the oldest token for which* $d|_j^i = c$, *where c is the token before b. If that token doesn't exist as well, it will investigate the previous one and so on.*

In the subtitling example, a token that contains some subtitle is transformed to the first frame on screen that shows this text. This frame can also be transformed back to the token that contains this text.

Tokens that cannot be transformed into each other via forward or backward transformations, might be transformed via another token. (E.g. in the subtitling example, a certain sentence in a subtitle corresponds to a certain number of frames in a video file. To transform this sentence's token, it has to be transformed to the final frame and then back to the video frame in the video file.)

Not only tokens can be transformed, moments can be transformed too:

Definition 3.3.8 *A moment τ in a certain time space T_i is transformed to another time space T_j by transforming the token with that time stamp in the channel in that time space:*

$$\tau|_j^i = t\left(u|_j^i\right), t(u) = \tau.$$

Some tokens influence more tokens than others; we say that some tokens have a bigger "impact" than other tokens. The number of tokens that is influenced, is called impact:

Definition 3.3.9 *The number of tokens $\mathbb{M}_j^i(a)$ from the first to the last token in channel c_j that are influenced by token a in channel c_i, $c_i > c_j$, is called the* impact *of token a in channel c_j.*

The impact of tokens doesn't need to be constant for all tokens in a certain channel. For example, an I frame will typically have a higher impact than a P frame. If one I frame is used to compute ten final frames, while one P frame is only used to compute one final frame, then the impact of one I frame is ten times the impact of one P frame.

The influence of a token is the set of tokens that is influenced. For example, the influence of a certain subtitling sentence could be the number or frames that the text will appear in.

Definition 3.3.10 *The* influence $\mathbb{I}_j^i(a)$ *in a certain channel c_j of a certain token a in channel c_i, $c_i > c_j$, is the set of tokens in channel c_j that is influenced by a:*

$$\mathbb{I}_j^i(a) = \left\{b | t(b) \in \left[t\left(a|_j^i\right), t\left(a|_j^i\right) + \mathbb{M}_j^i(a) - 1\right]\right\}.$$

The source is the opposite: the source of a token represents the earlier tokens that have influenced that token:

Definition 3.3.11 *The* source $\mathbb{S}_i^j(a)$, $c_i > c_j$ *of a token a in channel c_j, is the set of tokens in channel c_i that influence a:*

$$\mathbb{S}_i^j(a) = \left\{b \in c_i \mid a \in \mathbb{I}_j^i(b)\right\}.$$

The functions $\mathbb{M}_j^i(a)$, $\mathbb{I}_j^i(a)$, $a|_j^i$ and $\mathbb{S}_i^j(a)$ are called *transfer functions*. They are defined in such a way, that they tell exactly which tokens are influenced by other tokens. Those relations can be dependent on the contents of the tokens; however, that makes it impossible to compute the transformation functions on beforehand. Therefore, they are not allowed to be dependent of the contents of

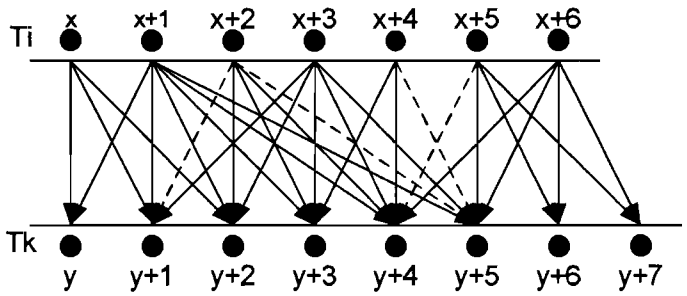

Figure 3.3: Some transformation characteristics of some arbitrary process. The solid lines represent the real influences, the dotted lines the parts that are added by the adapted functions.

the tokens. If they are, then the transformation functions cannot be defined, which places the processes in separate universes (universes are explained later). This is an important restriction of the model, as KPNs do allow this. Unfortunately, in such a situation, there is no way to predict these functions until the contents of the token are known, which is too late if tokens need to be transformed before they are processed.

It isn't always the case that a certain number of bits will, once processed, be a predictable number of bits. An example of this phenomenon can be seen in many compression algorithms. However, this has nothing to do with the predicted number of *tokens* that are output. What is important for synchronization, is in which tokens the information from certain tokens can be found further in the network, or, actually, which tokens are needed to produce a certain token. The *amount of information* or, consequently, the amount of bits that can be found in those tokens is a different question and in our case totally unimportant.

Now, it could for example be the case that information does not remain in the same order when it is processed: $t(a)|_j^i > (t(a)+1)|_j^i$ for some token a from channel $c_i > c_j$. This can be a drawback for certain calculations, like the influence calculation, that are looking for a set of tokens without knowing where the set ends. Therefore, a new function $\hat{\mathcal{I}}_j^i(a) = [\hat{A}(a), \hat{A}'(a)]$ is introduced, satisfying the following constraint of monotonicity:

$$a < b \Rightarrow \begin{cases} \hat{A}(a) \leq \hat{A}(b) \\ \hat{A}'(a) \leq \hat{A}'(b) \end{cases}$$

$\hat{A}(a)$ and $\hat{A}'(a)$ are altered versions of $\mathcal{I}_j^i(a) = [A(a), A'(a)]$, satisfying the following equations:

$$\hat{A}(a) = \begin{cases} A(a) & : \quad A(a) \leq \hat{A}(a+1) \\ \hat{A}(a+1) & : \quad A(a) > \hat{A}(a+1) \end{cases}$$
$$\hat{A}'(a) = \begin{cases} A'(a) & : \quad A'(a) \geq \hat{A}'(a-1) \\ \hat{A}'(a-1) & : \quad A'(a) < \hat{A}'(a-1) \end{cases}$$

and of course, $\hat{\mathbb{M}}_j^i(a) = |\hat{\mathcal{I}}_j^i(a)|$ and $\hat{a|_j^i} = \hat{A}(a)$. To compute $\hat{A}(a)$, one needs the past until $\hat{A}(a-1)$ and to compute $\hat{A}'(a)$, one needs the future from $\hat{A}'(a+1)$.

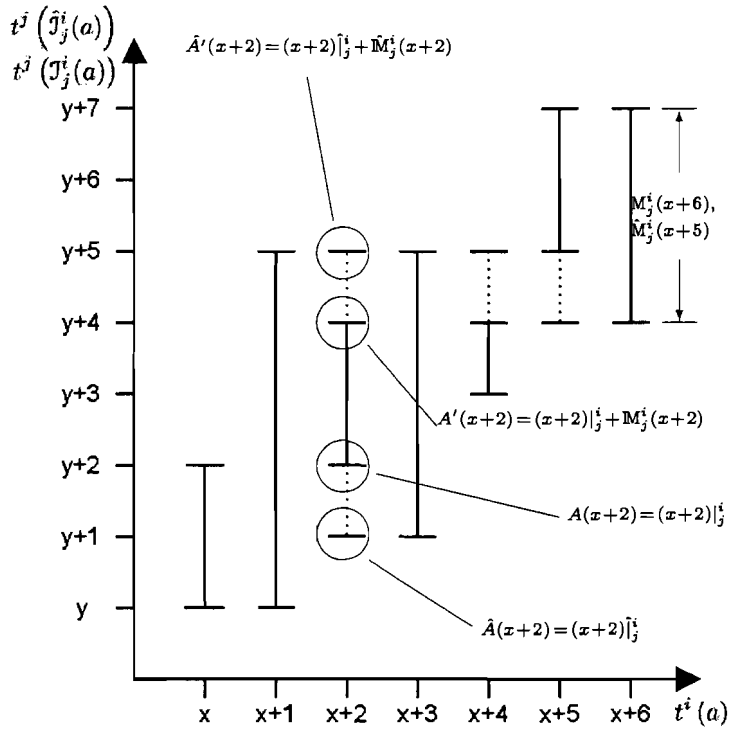

Figure 3.4: The influence of tokens from channel c_i in channel c_j, see also figure 3.3. The solid lines represent the influence $\mathcal{J}^i_j(a)$ of token a^i, the dotted lines and the solid lines together represent $\hat{\mathcal{J}}^i_j(a)$, like $\mathcal{J}^i_j(x+2) = [y+2, y+4]$ and $\hat{\mathcal{J}}^i_j(x+2) = [y+1, y+5]$.

In theory, one might really need the complete history, including the future to compute this $\hat{A}'(a)$, but usually, it will be possible to proof that only a few tokens are needed.

An illustration of these functions can be seen in figure 3.3 and 3.4. For tokens on moments $[x, x+6]$ the influences $\mathcal{I}_j^i(x, \dots)$ are shown in solid lines. In figure 3.3, the relations between time space T_i and T_j can be seen. The solid lines are the real influences. The dotted lines have been added in the altered functions. This isn't really clear in figure 3.4, but when this picture is converted to figure 3.4, this is much clearer: the envelope of the upper and lower bound are now monotonous. An advantage of this, is that if you know that a certain token (take for instance $y+7$) is not influenced by $x+4$, then you know as well that it isn't influenced by any token before $x+4$. The same is true the other way around: if you know that a token (take for instance $y+1$) is not influenced by $x+4$, then you also know that it isn't influenced by any token after $x+4$.

A function like $\hat{\mathcal{S}}_j^i(a)$ hasn't been defined, because that function would return the same set as $\mathcal{S}_j^i(a)$.

Take for example the token on $x+2$. One can immediately see that $(x+2)|_j^i = y+2$, $\mathbb{M}_j^i(x+2) = 3$, $\mathcal{I}_j^i(x+2) = [y+2, y+4]$, $\hat{\mathcal{I}}_j^i(x+2) = [y+1, y+5]$, et cetera. It is clear that with the dotted lines, the lower and upper envelope are now monotone.

3.3.4 Universes

As has been mentioned before, it is possible that transformation functions cannot be defined. In such a case, synchronization isn't possible, because for synchronization, it is necessary to know when information arrives at the different processes. To express this impossibility of synchronization, universes are introduced.

Definition 3.3.12 *Two time spaces T_1 and T_2, associated with two channels C_1 and C_2 respectively, $C_1 > C_2$, are synchronizable if $a|_2^1$, $a|_1^2$, $\mathbb{M}_2^1(a)$ and $\mathcal{I}_j^i(a)$ are defined $\forall a \in T_1$ and $\mathcal{S}_1^2(b)$ is defined $\forall b \in T_2$.*

Definition 3.3.13 *A universe is a set of synchronizable time spaces.*

Synchronization of processes is only possible if these processes are in the same universe. The opposite is true as well: processes that don't have to be synchronized don't have to be in the same time space and therefore don't need any transformation function or impact function. However, as transformation and impact calculations are performed while 'traveling' through the network graph, the time spaces that belong to the channels that connect two channels that have to be synchronized, *do* need to be in the same universe.

One can think of many situations in which these functions aren't defined. When for example the number of tokens that a process reads or writes depends on the contents of certain tokens, it can be impossible to define transfer functions. Another reason that these functions aren't defined, is simply that the network designer doesn't need them. If time spaces don't need to be synchronizable, then the existence of transfer functions isn't necessary.

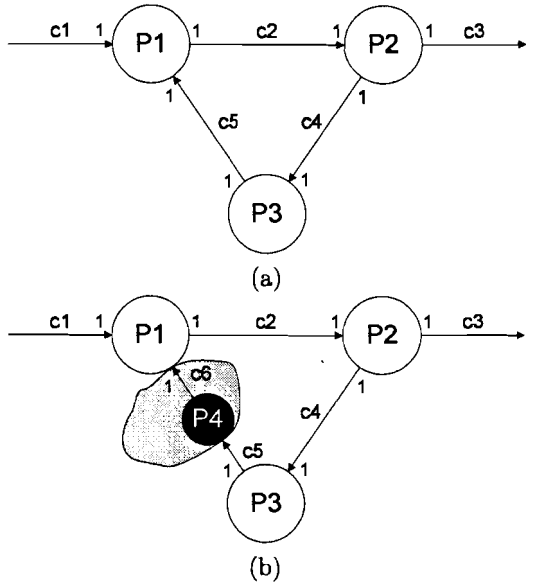

Figure 3.5: A network that contains a cycle. The numbers at the head and tail of the channels are the numbers of tokens that are produced or consumed per process loop. The grey area is a different universe from the one that the channels c_1, c_2, c_3, c_4, c_5 are in.

3.3.5 Cycles

As was said in paragraph 3.3.1, cycles in the network graph that is used to calculate distances and synchronization routes are only allowed under certain circumstances. However, many networks do contain cycles.

Consider the network in figure 3.5a. A token that comes at a certain time in channel c_1 will have an influence on a token in c_2, c_4, c_5 and eventually again in following tokens in c_2, c_4 and c_5, and so on. This means that the impulse of tokens that arrive at P_1 has to be infinite. So far so good.

We say, for example, that some token arrives in c_1 at moment t_1. The token that is then produced in c_2 will arrive at moment t_2 et cetera, and finally in c_5 at t_5. The token will now influence a token in c_2, that will arrive there some time later than t_2, say $t_2 + x$. That means that the token in channel c_2 at moment t_2 can be transformed to c_4 as t_4, to c_5 as t_5 and to c_2 as $t_2 + x$. So going round can transform a token in a certain time space to the *same* time space, but to another token. That is obviously wrong, though it is a theoretical problem, not a practical problem.

The source of this problem is the fact that when there are cycles in a network graph, there are infinitely many routes along which a transformation can be computed, which causes this ambiguity.

Cycles with this ambiguity are prohibited. Cycles in networks are always allowed, but in such a case, they should be in at least two different universes. Figure 3.5b illustrates the solution that is used: a dummy process P_4 is inserted with a channel c_6, that is placed in another universe. Transformations cannot

take place while crossing the borders of a universe, so only one route remains for each transformation. This way, every possible transformation is determinate and every real process can still be synchronized with any other process. There only remains one single route along which the transformations can take place.

In a case when there are no ambiguities however, cycles are allowed.

3.3.6 Subnetworks

Several connected channels and processes together can be treated like one big process.

A subnetwork has the same properties as a process and can be treated like one. If the network inside it is in one universe, then properties like the token impact and influence can easily be computed by expanding those functions to channels further in the network.

If $a|^i_j$ and $b|^j_k$ are known $\forall a, b$ then $a|^i_k = a|^i_j|^j_k$ is known too. If $\mathfrak{I}^i_j(a) = [A_1(a), A_2(a)]$ and $\mathfrak{I}^j_k(b) = [B_1(b), B_2(b)]$ are known $\forall a, b$, then $\mathfrak{I}^i_k(a) = [B_1(A_1(a)), B_2(A_2(a))]$ is known $\forall a$ as well, as is $\mathbb{M}^i_k(a) = |\mathfrak{I}^i_k(a)|$. This way, all functions can be computed from the subnetwork's incoming channels to its outgoing channels.

3.4 Reactive Networks

A reactive network is a network that reacts to non deterministic events that control the network functionality, which means that certain parts of the network change, after some event has taken place. This will normally be as soon as possible after this event occurred. When an event occurs, the network has to decide what its reaction will be, i.e. which processes have to change and how they should change. This has to be decided by the network, following the guidelines of the network designer.

Definition 3.4.1 *An* event *is a command, coming from the outside world (usually some kind of interface) at a moment that cannot be predicted a priori by the network, that tells the network that it should change its behavior, and how it should change this.*

Obviously, all these changing processes need to be in the same universe to be able to change in a synchronized way. If they're not, then a synchronized reaction is impossible. It is up to the designer of the network to decide whether to make time spaces synchronizable. Usually, many time spaces can be made synchronizable, though the often don't need to be, and it is easier then to leave them unsynchronizable. Unsynchronizable time spaces can still react to events, but not in a synchronized way.

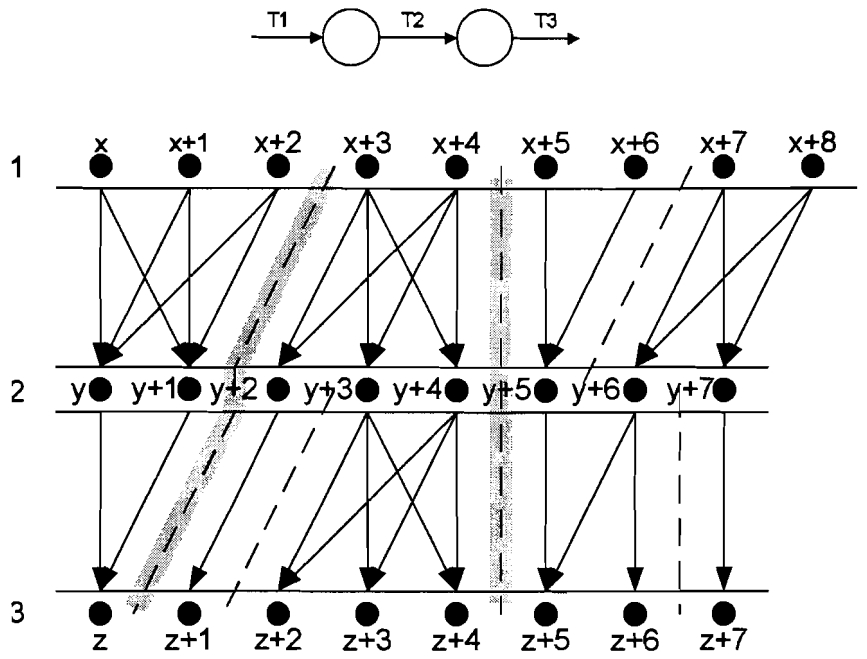

Figure 3.6: An illustration of the information dependencies of some stateless process. The dotted lines represent quiescent states. The grey areas are the quiescent states over several stages.

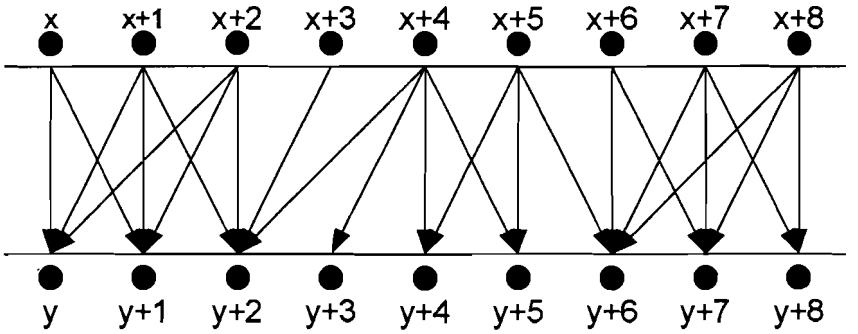

Figure 3.7: An illustration of the information dependencies of some stateful process. There are no quiescent states.

3.4.1 Stateless Changes

Stateless changes, as opposed to stateful changes, which will be treated in paragraph 3.4.2, are changes that are just carried out at a suitable moment and where only the processes are involved that were designated to change. The only constraints are that the changing processes all have to be in a *quiescent state* at the moment at which they'll change, and that that moment comes as soon as possible.

Is is quite obvious that one cannot go back in time. Tokens that have been processed, cannot be recovered. Therefore, the youngest token that has been consumed by one of the designated processes imposes the limit, before which no moment can be chosen at which the reconfiguration should take place. Starting with this moment, a moment has to be found at which every process that is involved will come into a quiescent state.

To find the youngest token that has been consumed, all the incoming channels of all the changing processes have to be searched. The token that is to be consumed first in each channel is the eldest token available, so the token at one moment earlier is the youngest token that has been consumed. Their time stamps have to be transformed into the same time space, after which they can be compared.

Every changing process should be in a quiescent state when the reconfiguration takes place and this moment should – transformed to their time spaces – be the same moment for all processes. This fact can be used to check whether a chosen moment is a quiescent moment. If there is a set of time spaces $\hat{T} = \{T_i | 1 \leq i \leq N\}$ that contains all time spaces that correspond to the input channels of the designated processes, then $\tau \in T_i$ is a quiescent state if $\tau|_j^i|_i^j = \tau, \forall_j$. This means that transforming to and back from all other time spaces in $\hat{T}$ gives the same moment. If not, one moment later can be tried, and this process continues until a moment is found that fits all processes. If no moment can be found, then there is no mutual quiescent state and synchronized reconfiguration is not possible.

This system is proven by examining definition 3.3.7. Some token a in time space 1 is transformed to time space 2, to the first token $b = a|_2^1$ that it influences. The back transformation $c = b|_1^2$ is to the first token that influences token b. If that is the same token ($a = c$) then no token before token a influences token b, so no information is used from tokens before that token.

This can be illustrated with figure 3.6. Suppose moment $y + 3$ is chosen. According to definition 3.3.8, $(y+3)|_i^j = x+3$ and $(x+3)|_j^i = y+2$, so $y+3$ is not a quiescent state. $y + 4$ would also transform to $x + 3$, which would transform to $y + 2$; wrong either. $(y + 5)|_i^j = x + 5$ and $(x + 5)|_j^i = y + 5$, so $y + 5$ is the first quiescent state, which is true, so $y + 5 \equiv x + 5$ is the first quiescent state available. The same process can be used to find quiescent points for different processes. The figure gives an illustration of two processes after each other. The grey areas denote the quiescent states for both processes behind each other.

Each process is then given the command to make its change happen at that moment, which means that it has to continue working until it has read all tokens with that time stamp, after which it should finish its work, output the

result and then start executing the new behavior.

The details on how to change are filled in by the designer, who will have to build some interface to communicate with the processes.

3.4.2 Stateful changes

The above principle works when all processes are stateless. However, it doesn't when stateful processes are involved, as will now be explained.

The question that is actually posed when one asks at which moment the processes should change, is *at which moment the 'old' way should be abandoned and the output should be computed the 'new' way.* It is this moment that determines when the processes should change.

Suppose that there is some network with some stateful process P somewhere in it. Suppose that somewhere in the path to that P's input stream there is a number of processes that change at a certain moment τ (which is of course properly transformed to each time space as applicable). Now the tokens arriving before τ are said to have been processed the 'old' way and tokens arriving from τ on are said to have been processed the 'new' way. As P is a stateful process, at moment τ and perhaps much longer, P will possibly be using 'old' and 'new' tokens together to compute output tokens. One can think of cases in which this is desirable or unimportant, but one can also think of cases in which this is highly undesirable or even just plainly wrong. Whether this is the case or not, should be defined by the network designer. It will usually depend on the kind of event or, more specifically, the difference between old and new tokens.

In the declaration of the event, the network designer can dictate which processes should be taken into account when the event happens. Now a graph is created, which is a subgraph of the network graph, containing all processes that will change, all processes that should be taken into account and all processes that lie in between. All these processes need to be replaced in order to be able to provide the last process with enough 'old' tokens to enable it to compute its last 'old' output token, and at the same time provide that process' new version with enough 'new' tokens to enable it to compute its first 'new' output token.

Example 2 *Consider figure 3.8. There is some stateful process P, before which several processes change at moment τ. The changing processes' input channel is in time space T_i, P's input channel is in time space T_j and P's output channel is in time space T_k. The tokens that P needs on its input channel to produce the last 'old' token at $\tau - 1$ are $S_i^j(\tau - 1)$. These tokens should be 'old' tokens. The tokens that P needs on its input channel to produce the first 'new' token at τ are $S_i^j(\tau)$. These sets will probably overlap, indicating that both an 'old' and a 'new' version of the same token are needed. An illustration of this can be seen in figure 3.9. It can be clearly seen that in this situation, there are tokens that are both needed to compute token $\tau - 1$ and token τ.*

To tackle this problem, two tracks are created, one along which the old tokens are computed and one along which the new tokens are computed. See figure 3.11 for an illustration. Track A will be the old track with the old processes that will keep running until the last token before the reconfiguration has been

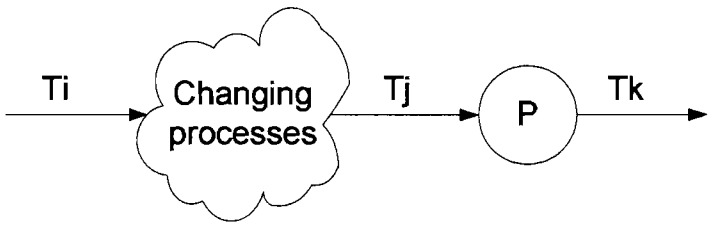

Figure 3.8: The network of example 2

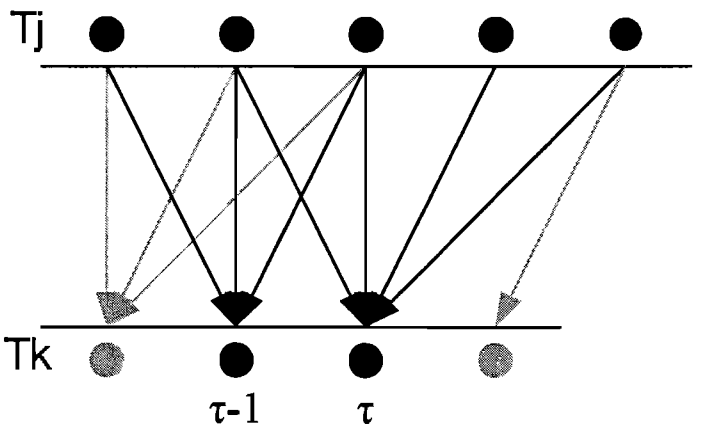

Figure 3.9: The source tokens of $\tau - 1$ and τ are overlapping.

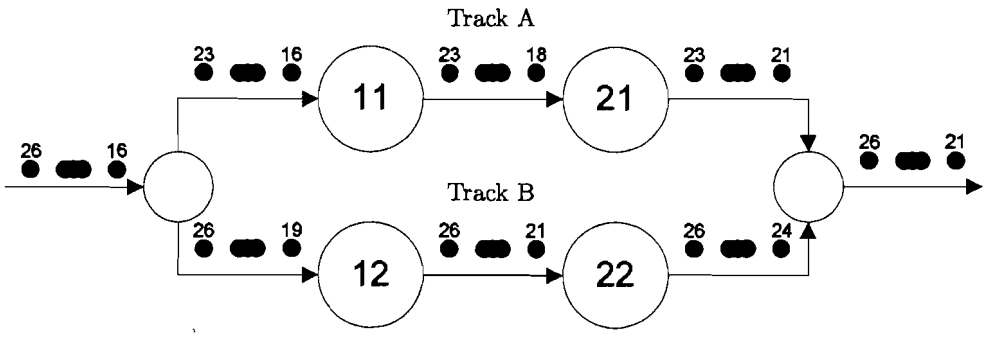

Figure 3.10: Stateful processes 11 and 21 have to be replaced by processes 12 and 22. The reconfiguration is scheduled at $\tau = 24$. $\mathcal{S}_1^3(23) = [16, 23]$ and $\mathcal{S}_1^3(24) = [26, 19]$. $\mathbb{M}_2^1(a) = 3, \forall_a$ and $\mathbb{M}_3^2(a) = 4, \forall_a$. Two tracks are created to compute two versions of the tokens that are needed by both processes.

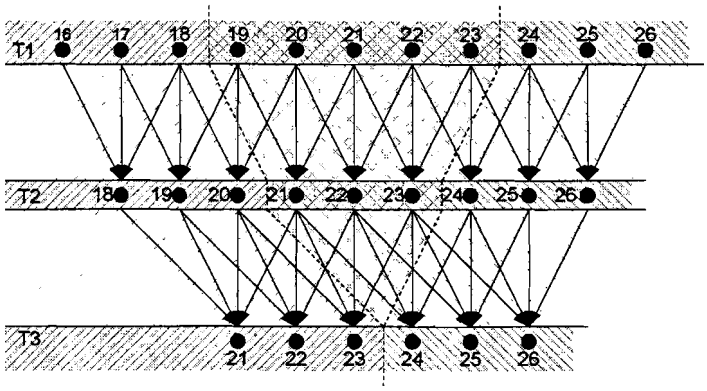

Figure 3.11: The influence relations from the example of figure . The overlapping areas have been shaded twice.

processed. Track B will be the new track, with all new processes. Track A will receive tokens until the last token of $\mathcal{S}_i^j(\tau - 1)$ and Track B will receive tokens starting with the first token of $\mathcal{S}_i^j(\tau)$. Part of this will overlap; those tokens are given to both track A and track B. When track A has produced its last token, the processes are terminated. Then, the tokens from track B will be output and the network will return to its normal state, but with new processes.

3.4.3 More Radical Changes

Thus far, only simple replacements of processes have been discussed. Of course, more radical changes than that might me desired. But whatever change is desired, the changing network can always be seen as a subnetwork, acting like a process that can be replaced. In that way, every possible change can be carried out.

3.5 'Rather synchronized' reconfiguration

In many situations, the transfer functions can be defined. However, there are also many situation in which these functions cannot be defined. In those cases, real synchronization is impossible, but there is an alternative which could work in many situations. Take for example a movie played in a window on a pc. Suppose that the movie is subtitled. Now if the window is resized, the movie should be resized too, and the subtitles should be resized at the same time. If that isn't possible, it could be a bit ugly if the subtitles would be resized a bit earlier or later than the movie itself. However, this wouldn't be a big problem. It's ugly, but that's all.

Definition 3.5.1 *The function $\mathcal{Q}(a)$ indicates whether a's time stamp is a quiescent state of the process that the channel that a is in feeds:*

$$\mathcal{Q}(a) = \begin{cases} \text{true} & : \ a \text{ is a quiescent state} \\ \text{false} & : \ a \text{ is not a quiescent state} \end{cases}$$

This function can be defined when it is impossible to create transformation functions, but when the quiescent states are known. Processes can then be changed, though they cannot be synchronized.

To tackle this last problem in certain situations, the function $\mathcal{C}_j^i(a)$ is introduced. This function has no true definition, but what it does can nevertheless be indicated:

If $a|_j^i$ is defined, then $\mathcal{C}_j^i(a) = a|_j^i$. If it isn't, then $\mathcal{C}_j^i(a)$ returns a moment from channel c_j that is 'close' to the moment that $a|_j^i$ had returned if it had existed. The implementation is left to the network designer.

Of course, 'close to' cannot be defined. It should however be clear what it means: as close to as possible under the current circumstances. The subtitling example should illustrate the idea.

$\mathcal{C}_j^i(a)$ can be used to make transformations 'rather synchronized'. In many applications, real synchronization isn't that important. It could however be

important to be as close as possible. In video for example, one or two distorted frames are undesired, but better than having no other option than starting the whole movie over again with different parameters.

3.6 Reconfiguration Practice

Several kinds of reconfiguration are possible, each having its own required functions:

1. No reconfiguration (the 'old-fashioned' KPN)
2. Asynchronous reconfiguration (requires $\Omega(a)$)
3. Rather synchronous stateless reconfiguration (requires $\Omega(a)$ and $\mathcal{C}_j^i(a)$)
4. Synchronous stateless reconfiguration (requires $a|_j^i$)
5. Synchronous stateful reconfiguration (requires $\mathcal{I}_j^i(a)$ (and $\mathcal{S}_i^j(a)$))

An implementation of this model ideally provides each of these reconfiguration possibilities, to be applied in the appropriate situation. The next section will describe the implementation that is proposed by the author.

3.7 Conclusion

The RPN model has been expanded to a model that makes KPNs reconfigurable. The model now is implementable. Important features of this model are the new perception of time, and the way that time progresses differently for each different time space. Reconfigurations take place synchronized, but in different time spaces, which means that the reconfigurations do not take place at the same moment when seen from the outside world.

Not all time spaces can necessarily be synchronized. Therefore, reconfigurations can take place in a synchronized way and in a non-synchronized way. Reconfigurations can be stateless and stateful. Non-synchronized reconfigurations can sometimes be 'rather' synchronized. These things are left to the network designer to decide.

Chapter 4

Implementation

4.1 Introduction

The theoretical model, that has been described in the previous chapter, has been implemented in YAPI, a tool box for C++ with which KPNs can be implemented. This chapter describes two things:

- How the theory was adjusted in order to make it implementable in YAPI.
- How this model was implemented in YAPI.

This chapter wasn't intended to be an implementation manual. It only described the way that things are organized in the implementation and why it is organized like that. A detailed how-to will be given in the next chapter.

4.2 Refinements

For several reasons, the implementation needed to make several small refinements to the theoretical concept, that was described in chapter 3. These refinements are the following:

1. The systems makes a difference between state changes and parameter changes, while the theory considers them equal.
2. Universes have not been implemented as such.
3. Rather synchronized transformations haven't been implemented as such either.
4. The influence and the source function have been slightly changed.
5. Stateful reconfiguration works differently.
6. Processes now explicitly execute loops.
7. The focus is now much more on process time than on channel time.

Ad 1: Parameters have been introduced for practical reasons. It wouldn't be a very good idea to implement a separate state for every possible combination of parameters. Strictly, every change in a process functionality remains a state change. There is no clear distinction between a parameter and a state. In theory both can be used to model every possible behavior. In practice, the difference is big: changing a state makes the process really execute another function, while a parameter change only changes one variable in the process.

Ad 2: In the theory, universes have been introduced to express the situation that processes cannot be synchronized. In the implementation, universes have not been implemented explicitly, because it's not necessary to be explicit about that. When two processes cannot be synchronized, then the programmer just shouldn't do that.

Ad 3: The same holds for rather synchronous reconfiguration. It is up to the programmer to make sure that the transformation functions are correctly implemented. When these functions are not accurate, then the synchronization is automatically 'rather' synchronous, instead of synchronous. The system doesn't need - and therefore doesn't make - any distinction between rather or real synchronization. It doesn't check whether or not the results are correct. Internally, there is no distinction between these transformations.

Ad 4: The influence function should give a set of tokens. That isn't very practical for programming. If it is known that the elements of a set are consecutive moments in time and the smallest element and the size of a set are known, then the influence is known as well. (See definition 3.3.10.) Something comparable has been done for the source function: only the size is given. When the size is added to the result of the back transformation, the answer is known.

Ad 5: Stateful reconfiguration is carried out with only one version of each process, not with two.

Ad 6: In fact, processes don't need to execute something that has the construction of a loop. In practice however, that will nearly be always the case. The implementation assumes that processes execute loops, because that is, given the construction of YAPI, the best way to be able to implement state changes. Its construction reflects that thought.

Ad 7: Time stamps are in principle something that has to do with channels. When a token appears in a channel determines which time stamp that token receives. However, these time stamps are used to make processes change. Therefore, it is the process that keeps track of the time of the relevant channels. (Another reason to do this, is that YAPI was constructed in such a way that it is nearly impossible for process objects to communicate with channel objects, other than through buffers.) This is done by keeping track of the number of tokens that have been read. Tokens are read in the same order as they're written, so this is a valid method to keep track of their time stamps.

In spite of these adjustments, the theory remains valid in the implementation. The only exception is the stateful reconfiguration; the structure of that has changed thoroughly.

4.3 Architecture

In order to implement the theory in YAPI, several things are needed:

1. Networks that can change
2. Processes that can change
3. A way to give the tokens a time stamp
4. A way to transform time stamps

To implement these, the YAPI `class Process` has been extended, by deriving a class from it, as well as the YAPI `class ProcessNetwork`. The derived classes are called `class RpnProcess` and `class ReactiveProcessNetwork`.

4.3.1 Time Stamps

The theory gives each token a time stamp, which is computed very easily: the first token in a channel is number 1, the second number 2 and so on. Every channel is only read by one process and the processes are the only instances in the network that really need the information about each token, so therefore, the tokens are just counted by the process that takes its input from the channel.

Following the theory, the time stamps of the tokens are used as a clocking mechanism, i.e. as a system that provides a way to express the moment that a process should change. The type `Time` is used to express all variables that represent some moment, which is of the same type as the type that YAPI uses to count the number of tokens that it has processed. (Therefore, the possibility of overflowing hasn't been taken into account; this would require a much more complicated system, which would only be needed for very large numbers of tokens.)

So the task of keeping track of the time stamps of tokens is designated to the processes, not to the channels. The reason that this was done that way, is that the design of YAPI is such that it is nearly impossible to have process objects communicate with fifo objects, other than through reading and writing.

4.3.2 Processes

So the YAPI processes had to be changed in two ways:

1. Processes needed timing functionality.
2. Processes needed reconfiguration functionality.

`RpnProcess` has got a variable `Time clock` (`Time` is the type in which time is expressed, defined in `time.h`) that keeps track of the process's clock. The behaviour of a process is illustrated in figure 4.1. After the initialization, the process enters a loop, in which the clock is adjusted (usually according to the number of tokens read from some input channel), the correct state is selected and

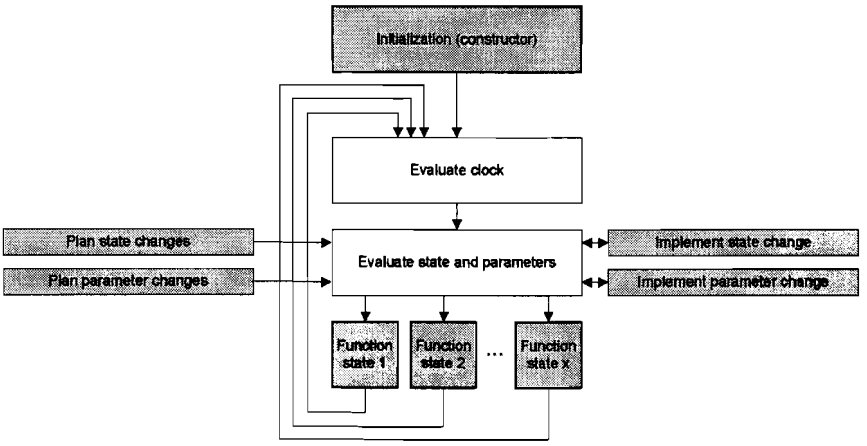

Figure 4.1: The internal structure of a running process. The grey processes can be redefined by the programmer. The thickly lined processes *have* to be defined by the programmer.

the corresponding function is called. These functions (usually in the format `Time main1()`) are written by the programmer. Such a function returns the number of moments that the clock has increased during this process, i.e. the number of tokens that have been read from a certain channel during the execution of this function. As is described in figure 4.1, not all processes need to be written by the programmer. Most processes have a standard implementation with a `virtual` function, which can be re-implemented by the programmer. Details can be found in chapter 5.

A process can only be reconfigured after such a process function has finished. If the process doesn't save any information, computed with information from tokens, when such a function ends, then those states are the quiescent states, mentioned in the theory. This is the case for stateless processes. The occurrence of quiescent states has to be predicted by the programmer. He is expected to provide a function `bool Q(const Time moment)` which returns `true` if `moment` is a quiescent state (see section 3.5). This function is used to produce the results of functions like `NextQ()`, which returns the next available quiescent state by checking which moment, starting with the current value of `clock` makes `Q(const Time moment)` return `true`. The maximum number of moments that `NextQ()` searches, is defined by the constant `const Time MAXSEARCH = 100;` in `rpnprocess.h`.

Internally, the process has a list of planned changes, which is filled by calling `void ChangeState(int s[, Time moment])` or `<template T> void ChangeParam(int param, T newValue[, Time moment])`. The process stores its state in the variable `state` and updates it according to this list. Whenever a state is updated, the process calls `void ImplementStateChange(const int newState)`.

For parameters, a comparable structure is applied, but there is one major difference: `RpnProcess` itself stores nor changes the parameters, because they have to be defined by the programmer and because they can be of any type, and because `RpnProcess` doesn't know which type it has, it cannot access those parameters .

Therefore, in `param.h` `class ParamChange` is declared, which is, together with its derived `class TParamChange` used to plan a parameter change. The details will be explained in the next chapter (section 5.2).

A function `Time M(Time moment[, int i[, int j]])` is provided, which should give the impact $\mathbb{M}_j^i(moment)$ of a certain token. If the programmer doesn't re-implement this function, it just returns `1`. There is no function for the influence, but it can easily be computed as described in 3.3.10.

The same holds for the source function. No set is given, but the first element and the size. The first element is given by firstelement($\mathbb{S}_i^j(moment)$)=`Time S(Time moment[, int j])`. The size $|\mathbb{S}_i^j(moment)|$=`Time Ss(Tme moment[, int j])`.

4.3.3 Subnetworks

A subnetwork should behave like a process, so basically has the same functions as a process. However, a subnetwork doesn't have a clock. When it is asked for its clock, it transforms the clocks of all processes to the same time space (Time space "0") and returns the highest. Which time space this time space "0" is, is a design choice, that has to be made by the programmer. Usually the time space of some incoming channel will do, but the programmer could for example also decide to create a new time space for that. To check the clock of each process, a network needs access to all (relevant) processes and process networks that it contains. It therefore has a list of both. (`vector<RpnProcess*> processes` and `vector<ReactiveProcessNetwork*> rpns`) The programmer needs to add all relevant processes and process networks to these lists.

A major difference between a process and a process network, is that the process network doesn't plan the reconfigurations to be executed at some later point in time. It just orders its processes to plan the reconfiguration to be executed at the chosen moment. This way, a reconfiguration can be planned by invoking the correct function of the top-most network. This network then can invoke the correct functions of the appropriate subnetworks and processes, each with the correctly transformed (see section 4.3.4) moment. This way, the processes will each change at the appropriate time.

4.3.4 Transformations

Transformations are a difficult issue in the implementation of reactive process networks. The program cannot deduce the transformation functions by itself. It is up to the programmer to create correct transformation functions. This is a drawback of KPNs. In KPNs, everything with respect to reading and writing tokens is allowed. In other streaming models, like SDF, this isn't the case and in such a case, the transfer functions can be detected by the system, by tracing how many tokens a process reads and writes per loop. With KPNs, the programmer has to write them himself. However, he is assisted by the architecture of the system.

Processes and process networks have transformation functions `Time TransformIJ(Time moment[, int i[, int j]])` and `Time`

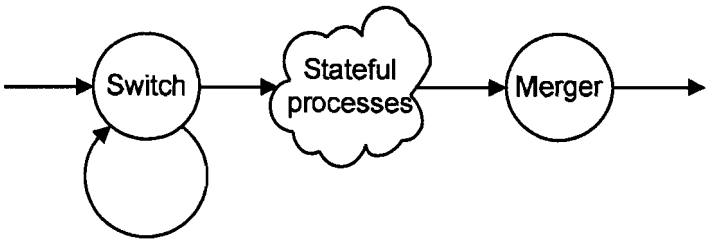

Figure 4.2: The structure of stateful reconfiguration.

`TransformJI(Time moment[, int i[, int j]])`, which can be used to redefine the transformation functions $moment|_j^i$ and $moment|_i^j$. They can (and probably should) be dependent on the state and parameters of the process, and of the planned changes. Examples can be found in the next chapter.

Process networks have transformation functions that transform a moment for a certain process (which is in fact a time space that is connected to the process) to a general time space. (`TrOI` and `TrIO`, the `O` is a zero, not an O) Which time space this is, is a design choice. Again, this generally will depend on things like the network structure and the states and parameters of the different processes, so this has to be done by the programmer. These functions are used to synchronize time spaces in order to be able to compare them.

4.3.5 Stateful Reconfigurations

Stateful reconfigurations can be quite complicated. Therefore, it is hard to talk about them in a general way. The principle however, is always the same.

In figure 4.2, one can see how stateful reconfiguration is performed. The theory uses two separate processes, which is clearer in theory. In practice, it is easier to just use one process. The main reason to do that, is that it simplifies the network that is used. Another reason to do this, is that one doesn't have to keep track of which process is active at which time. A third reason to do this, is that YAPI is constructed in such a way, that inactive processes are always either reading or writing, because a process just continues executing its main function, until a read or a write operation cannot be completed because a buffer is full or empty. The process that should start in another state cannot switch to this state at that moment, because it was already executing a main function that belongs to some previous state. Therefore, it cannot switch immediately, because it can only switch when a main function returns.

Figure 4.2 shows part of a process network, namely the part that should be part of the switch. The input channels have a stateful switch (`SFS` in `sfs.h`) and the output channels have a stateful merger (`SFM` in `sfm.h`).

The switch continues until it arrives at the moment that tokens need to go to both processes, as seen in the theory. (Figure 3.11) From that moment on, tokens are still sent to the process, but are also saved in the buffer (the loop in figure 4.2) to be used later. At the moment that the old version has finished what it should do, it makes sure that the process switches and it empties this

buffer, so that the tokens are sent again to the process network, which then uses them again to compute what it has to compute, but now in the new state. When the buffer is emptied, the switch continues its work by again just passing tokens from the input to the output.

The merger does the opposite: it ignores the tokens that are wrong. (It could for example be the case that a process always outputs two tokens at a time, while the reconfiguration should be between them, or that a process, due to its internal structure always starts with some noise.) This is very much like the theory. The process continues reading tokens from the input and writing tokens to the output, until the process has switched. At that moment, the process starts to ignore the tokens that have wrong results (if there are any) until the first correct token arrives. At that moment it switches back and continues reading and writing the way it used to do.

This system (like the theoretical system by the way) has to be customized for every set of stateful processes, because the system itself has no information about the network structure and where impacts influence each other - this depends on the network architecture. An example which implements this, is shown in the next chapter, which describes the implementation.

Chapter 5

Manual

5.1 Introduction

This manual explains how the RPN-toolbox is used to write RPN-programs. It is assumed that the reader already has a thorough knowledge of YAPI. This can be found in [7]

To illustrate how to use the extended version of YAPI, a small RPN-program is made step by step. The program that has been chosen is a small filter, which can be seen in figure 5.1. This filter can be seen as a model for e.g. a volume control. It consists of two separate reactive processes: a duplicator and a multiplier. The duplicator has two states. In state 1 it is turned off and just passes the token to the next process. In state 2 it duplicates the tokens that it reads n times. n is provided as a parameter.

The multiplier only has one state, but it has a parameter, that it multiplies its input with.

The source provides tokens in some way and sends them to the filter. The sink reads them and sends them to their destination, whatever that may be. The source and the sink are normal KPN processes and therefore, they are not treated in this manual. There is no problem to use KPN processes mixed with

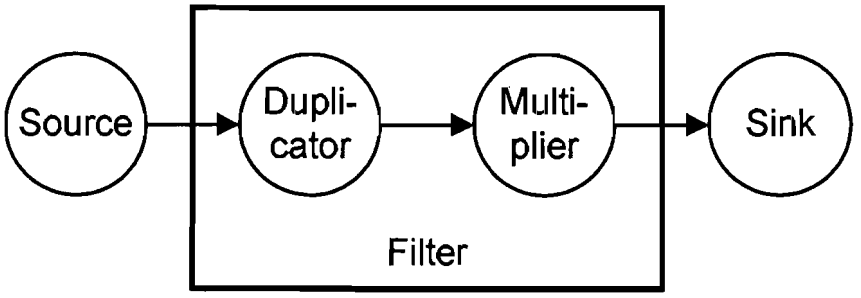

Figure 5.1: The schematics of the filter program.

RPN processes.

5.2 The RpnProcess: The Multiplier

Program Text 1: multiplier.h

```
#include "rpnprocess.h"

class Multiplier : public RpnProcess
{
public:
  Multiplier(const Id& n,
             In<int>& i,
             Out<int>& o,
             int f=1);
  const char* type() const;

  bool Q(const Time moment);

  Time main1();

private:
InPort<int> in;
OutPort<int> out;
  int factor;

  void ImplementParamChange(ParamChange* p);
};
```

The header file of the multiplier can be seen in program text 1. Most of this is equivalent to how things are done for normal processes. There are a few exceptions:

- Each `RpnProcess` must have a member function `bool Q(const Time moment)`. This function returns whether or not `moment` is a quiescent state. For example, it could be the case that each even number is a quiescent state.
- A parameter can be adjusted, so a member function `void ImplementParamChange(ParamChange* p)` is needed.
- Main functions are in the format `Time <name>()`.

Program Text 2: multiplier.cc

```
#include "multiplier.h"

Multiplier::Multiplier(const Id& n, In<int>& i,
                       Out<int>& o, int f):
  in(id("in"), i),
  out(id("out"), o),
  factor(f),
```

```
  RpnProcess(n, 1)
{
  states[1]=(MAIN)&Multiplier::main1;
}

const char* Multiplier::type() const {return "Multiplier";}

bool Multiplier::Q(const Time moment) {return true;}

void Multiplier::ImplementParamChange(ParamChange* p)
{
  factor=((TParamChange<int>*)p)->NewValue();
}

Time Multiplier::main1()
{
  int i;
  read(in, i);
  write(out, factor*i);
  return 1;
}
```

Program text 2 gives the implementation of `Multiplier`. The first notable thing one can see is the initialization of its ancestor, `RpnProcess`, in the constructor. The second argument represents the initial state of the process. This is 1 by default, but can be specified differently.

The second thing is the initialization of the different states in the constructor. This is done by adding each main function to the vector `states`. These functions need to be converted to the type `MAIN` (which is defined in `rpnprocess.h` by `typedef Time (RpnProcess::*MAIN)();`), because `RpnProcess` expects functions that are of type `Time (RpnProcess::*)()`. This is dictated by C++. `Time main1()` is the main function, so state 1 is initialized with `states[1]=(MAIN)&Multiplier::main1;`. This main function basically does four things: it reads a token, it multiplies it with the current multiplying factor, it outputs it and it returns the number of moments that it needs, in this case 1.

The Q function in this case just returns `true` because every moment is a quiescent state.

The parameter changing function `ImplementParamChange(ParamChange* p)` is called at the moment that a change really should take place (so perhaps some time after this change was planned). This always happens before the new main function is called. The only thing it does in this case, is to give `factor` a new value. This new factor is given in parameter `ParamChange* p`. This however isn't really trivial. The problem actually is that the super class `RpnProcess` doesn't know which parameter types its childs have. Therefore, it passes the parameter as type `ParamChange*`. This type has two member functions, `int ParamNr()`, returning the parameter number (i.e. 1 for parameter 1, 2 for parameter 2 et cetera) and `Time Moment()`, which returns the moment on which the change should take place, which is of course equal to the clock when `ImplementParamchange` is called. We know that parameter number 1 (there is only one parameter) is an integer, so we also know that the actual type of the

parameter is `TParamChange<int>*`. Therefore, we need to convert this parameter to that type, so we can call its member function `T NewValue()` where `T` is an `int`. Therefore, `factor=((TParamChange<int>*)p)->NewValue();`.

The rest of the .cc file is just the same as it would normally be with YAPI.

5.3 The RpnProcess: The Duplicator

Program Text 3: duplicator.h

```
class Duplicator : public RpnProcess
{
public:
  Duplicator(const Id& n,In<int>& i,Out<int>& o,
             int initialState=1,int number=1);
  const char* type() const;

  inline int NumberOfDupl() {return numberOfDupl;};

  bool Q(const Time moment);
  Time TransformIJ(const Time moment,const int i=1,const int j=1);
  Time TransformJI(const Time moment,const int j=1,const int i=1);

  void ChangeState(int s, Time moment);
  void ChangeState(int s);
  void ChangeParam(int param, int newValue, Time moment);
  void ChangeParam(int param, int newValue);

  Time main1();
  Time main2();

private:
  InPort<int> in;
  OutPort<int> out;
  int numberOfDupl;

  void ImplementStateChange(const int newState);
  void ImplementParamChange(ParamChange* p);

  Transform transformations;
};
```

When it comes to the .h file, there's not so much difference with the multiplier, except that there are two main functions (two states) and that transformation functions have been added as well as a transformation variable. These are explained below.

Program Text 4: duplicator.cc

```
Duplicator::Duplicator(const Id& n,In<int>& i,Out<int>& o,
                       int initialState,int number):
  in(id("in"), i),
```

```
  out(id("out"), o),
  numberOfDupl(number),
  RpnProcess(n, initialState)
{
  states[1]=(MAIN)&Duplicator::main1;
  states[2]=(MAIN)&Duplicator::main2;
  if (initialState==2) transformations.ChangeRate(1, 1, number);
}

const char* Duplicator::type() const {return "Duplicator";}

bool Duplicator::Q(const Time moment) {return true;}

Time Duplicator::TransformIJ(const Time moment,const int i,
                             const int j)
{ return transformations.TransformIJ(moment);}

Time Duplicator::TransformJI(const Time moment,const int j,
                             const int i)
{ return transformations.TransformJI(moment);}

void Duplicator::ChangeState(int s, Time moment)
{
  StatelessProcess::ChangeState(s, moment);
  // Register clock
  if (s==1) transformations.ChangeRate(moment, 1, 1);
  if (s==2)
    transformations.ChangeRate(moment, 1,
      ((TParamChange<int>*)ParamAt(moment))?
      ((TParamChange<int>*)ParamAt(moment))->NewValue():
      numberOfDupl);
}

void Duplicator::ChangeState(int s){ChangeState(s, NextQ());}

void Duplicator::ChangeParam(int param,int newValue,Time moment)
{
  StatelessProcess::ChangeParam(param, newValue, moment);
  // Register clock
  if (State()==2) transformations.ChangeRate(moment, 1, newValue);
}

void Duplicator::ChangeParam(int param, int newValue)
{
  ChangeParam(param, newValue, NextQ());
}

Time Duplicator::main1()
{
  int t;
  read(in, t);
  write(out, t);
  return 1;
}
```

```
Time Duplicator::main2()
{
  int t;
  read(in, t);
  for (int i=1; i<=numberOfDupl; ++i) write(out, t);
  return 1;
}

void Duplicator::ImplementStateChange(const int newState)
{ // Nothing is needed, the state changes automatically...}

void Duplicator::ImplementParamChange(ParamChange* p)
{
  numberOfDupl=((TParamChange<int>*)p)->NewValue();
}
```

Now it is clear that the duplicator is a bit more complicated than the multiplier. These differences can be divided into two different categories:

1. There are two states
2. There are non-standard transformation functions.

5.3.1 More than one state

It is quite simple to give a process more than one state, as can be seen in the program text. The programmer only needs to define one main function for each state and then add them to the vector `state`. The process will automatically choose the appropriate main function. A state change doesn't really need an implementation in this case, because apart from picking the other main function, nothing really needs to be altered.

5.3.2 Transformations

One challenge with this duplicator is that the transformation $\tau|_j^i$ can change, depending on the state and parameter of the process. Where the information of some input token will be in the next channel, depends not only on the present, but also on the planned state changes. However, this information is needed to plan the changes for the processes after this process.

Therefore, every planned change in input/output rate is registered as soon as it is planned. To do this, the process reimplements the functions with which state and parameter changes are planned. In the program text, one can see that first `RpnProcess`'s functions are called. Then the planned transformation changes are registered.

These transformations are registered in the variable `transformations` of type `Transform` by calling `void Transform::ChangeRate(Time moment, int i, int j)`. `moment` is the moment that the change takes place, `i` is the number of tokens read per time period and `j` is the number of tokens written per

time period (usually per loop, but it could easily be per two loops, or per twenty loops or so). `Time Transform::TransformIJ(Time moment)` and `Time Transform::TransformJI(Time moment)` give the forward and backward transformations $moment|_j^i$ and $moment|_i^j$, by calculating them while making use of the registered transformations. (This only works for this specific kind of behavior, i.e. a constant number of input and output tokens per time period between two reconfigurations. In other cases, the programmer will need to develop his own method to compute these transformations.)

The registering process depends on the state that is planned. If state 1 is planned, then of course the input and output rates are just one. If state 2 is planned, then they depend on the value of the parameter at that moment. This is tested by using the function `RpnProcess::ParamAt(Time moment)`, which gives the last parameter change before `moment`. (As there's only one parameter, one doesn't need to look for a specific parameter number.)

The transformation functions themselves make use of `transformations` to calculate their transformations.

5.4 The ReactiveProcessNetwork

The filter itself is a `ReactiveProcessNetwork`, which is derived from `class ProcessNetwork`. It doesn't have more than one state, but it has two parameters: the multiplication factor and the duplication factor. They each have a number that is defined in a const: `P_FACTOR` and `P_DUPL` . The header file can be seen in program text 5.

Program Text 5: filter.h

```
const int P_FACTOR=1;  // Parameters numbers, to be used
const int P_DUPL=2;    // to identify them.

class Filter : public ReactiveProcessNetwork
{
public:
  Filter(const Id& n, In<int>& i, Out<int>& o, int factor=1);
  const char* type() const;

  Time TrIO(RpnProcess* r, Time moment);
  Time TrOI(RpnProcess* r, Time moment);

private:
  Fifo<int> f;

  InPort<int> in;
  OutPort<int> out;
  Multiplier m;
  Duplicator<int> d;

  void ImplementParamChange(ParamChange* p);
};
```

Most of the file should be clear. Two things are different: the parameter change implementations and the transformation functions, which are explained in section 4.3.4.

Program Text 6: filter.cc

```
#include "filter.h"

Filter::Filter(const Id& n, In<int>& i, Out<int>& o, int factor):
  f(id("f1")),
  in(id("in"), i),
  out(id("out"), o),
  m(id("m"), f, out, factor),
  d(id("d"), in, f, 2, 1),
  ReactiveProcessNetwork(n)
{ }

const char* Filter::type() const{return "Filter";}

Time Filter::TrIO(RpnProcess* r, Time moment)
{
  if (r==&m) return d.TransformJI(moment);
  return moment;
}

Time Filter::TrOI(RpnProcess* r, Time moment)
{
  if (r==&m) return d.TransformIJ(moment);
  return moment;
}

void Filter::ImplementParamChange(ParamChange* p)
{
  switch (p->ParamNr()) {
  case P_FACTOR:
    int newValue=((TParamChange<int>*)p)->NewValue();
    m.ChangeParam(1,newValue,TrOI(&m, p->Moment()));
    break;
  case P_DUPL:
    int newValue=((TParamChange<int>*)p)->NewValue();
    if (newValue==1)
      d.ChangeState(1, p->Moment());
    else {
      d.ChangeState(2, p->Moment());
      d.ChangeParam(1, newValue, p->Moment());
    }
    break;
  }
}
```

The first interesting part of `filter.cc` is the transformation functions. They test for which process a transformation is needed, perform the corresponding transformation and return the answer. The input channel is considered as time

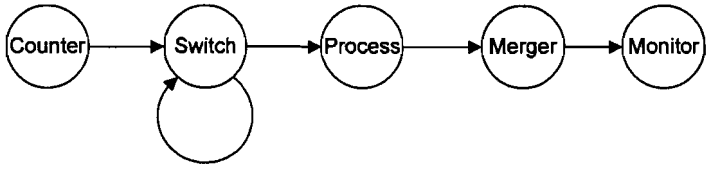

Figure 5.2: The schematics of the stateful illustration program.

space 0. According to the network topology, the incoming channel of the duplicator is in the same time space, so no extra transformations are needed for it. The multiplier is behind the duplicator, so transformations for that process and its incoming channel need to be transformed by `duplicator`'s transformation functions. That's why the function tests whether the desired process is the multiplier, and if so, it transforms `moment` with `d.TransformJI(moment)` or `d.TransformIJ(moment)`.

The other interesting thing is the parameter change implementation function. This function tests which parameter is to be changed in the `switch` statement. It then applies the corresponding transformation to the correct `template<class T>TParamChange<T>*` to read the new value. When that is finished, it reconfigures the correct process.

It is clear that the different RPNs are responsible for the correct implementation of the reconfigurations. The processes just wait until the network gives them the order to reconfigure.

5.5 Stateful reconfigurations

A small network has been made to illustrate stateful reconfiguration, see figure 5.5. In state 1, the process (called `process1` in this case) outputs the average of the last three tokens and saves the last two tokens. In state 2, the process outputs twice the average of the last three tokens and saves the last two tokens * 2. So $\mathcal{S}_i^j(\tau) = [\tau - 2, \tau]$ and $\mathcal{I}_j^i(\tau) = [\tau, \tau + 2]$. This process can be seen in program text 7 and 8 The interesting part of the program text is the switching function of the network, which can be seen in program text 9.

Program Text 7: testprocess.h

```
#include "rpnprocess.h"

class TestProcess : public RpnProcess
{
public:
  TestProcess(Id& n, In<int> i, Out<int> o);
  const char* type() {return "TestProcess";};

  Time M(const Time moment, int i=1){return 3;};
  Time S(const Time moment, int j=1){return moment-2;};
  Time Ss(const Time moment, int j=1){return 3;};
```

```
  Time main1(); // Average of the last three tokens
  Time main2(); // Average of the last three tokens times 2

private:
  InPort<int> in;
  OutPort<int> out;

  int token1, token2;
};
```

Program Text 8: testprocess.cc

```
#include "testprocess.h"

TestProcess::TestProcess(Id& n, In<int> i, Out<int> o):
  in(id("in"), i),out(id("out"), o),token1(0),token2(0),
  RpnProcess(n)
{
  states[1]=(MAIN)&TestProcess::main1;
  states[2]=(MAIN)&TestProcess::main2;
}

Time TestProcess::main1()
{
  int i;
  read(in, i);
  write(out, (token1+token2+i)/3);
  token1=token2;
  token2=i;
  return 1;
}

Time TestProcess::main2()
{
  int i;
  read(in, i);
  i*=2;
  write(out, (token1+token2+i)/3);
  token1=token2;
  token2=i;
  return 1;
}
```

Program Text 9: void ImplementStateChange

```
void StatefulNetwork::ImplementStateChange(const int newState,
                                           const Time moment)
{
  // To stay producing, the last token it needs is:
  Time m1=process1.S(moment-1)+process1.Ss(moment-1)-1;
  // So it should switch one token later:
  process1.ChangeState(newState, m1+1);
```

```
    // To produce tokens from moment, it needs tokens from S(moment).
    Time m2=process1.S(moment);
    sfs.SwitchState(m2, m1);

    // Tokens that are to be ignored are at moment, the number is:
    Time m3=moment-process1.S(moment);
    sfm.IgnoreAt(moment, m3);
}
```

The stateful process is `process1`, the stateful switch is `sfs` and the stateful merger is `sfm`. When a change is planned on moment `moment`, the process needs to stay producing until the last token before `moment`, which is `moment - 1`, so it needs input until the last token in $\mathcal{S}_i^j(moment-1)$, which is `process1.S(moment - 1) + procesi.Ss(moment - 1) - 1`[1]. The reconfigured process needs all tokens in $\mathcal{S}_i^j(moment)$, so it should start with token `process1.S(moment)`.

[1]For those functions, see section 4.3.2

Chapter 6

Conclusions and Recommendations

A model has been presented, that implements the outline of RPNs[5]. This model improves the KPN model [6], enabling it to model dynamic networks. The key principle to achieve this, is to use a different concept of time. Time shouldn't be divided into seconds, but into the sequence numbers of tokens. That means that the time isn't equal for each process and channel.

Every part of the network is considered to be in a different time space, each with its own clock. Those time spaces are related by the processes between the different channels. Crucial for these relations is which tokens' information correspond to each other in terms of information.

Several time spaces together can form a universe, inside which the time spaces can be synchronized. When time spaces can be synchronized, the processes can be synchronized too, and thus they can be reconfigured in a synchronized way.

There are several forms of synchronized and unsynchronized reconfiguration, see section 3.6 for an overview. For each way, the reconfiguration has been modeled.

The model has been implemented to be used by YAPI. A description has been given of the underlying principles. Not every part of the theory has been implemented exactly the way that the theory describes. These differences have been investigated and explained.

Several choices have had an impact on the implementation:

1. No attention was given to the fact that counters can overflow. This choice was made, because the model is not intended to simulate endlessly. In practice, the limits should do.
2. Providing correct transformation functions are the job of the programmer. This choice was made, because YAPI isn't really suitable to synthesize a network model, while this model is needed to calculate the transformations automatically.
3. In the implementation, it is the process that keeps track of the channel time. This was done this way, because YAPI is designed in such a way that

it is impossible for process objects to communicate with channel objects (except the tokens of course).

4. Processes now explicitly execute loops. This choice was made, because it was the best way to explicitly distinguish quiescent points (at the end of each loop), to switch between states and to keep track of which time stamps the just read tokens have.

A manual has been written, with which a programmer is able to write his own RPN programs.

There are several areas in which there is still work to be done.

1. The stateful reconfiguration is now quite complicated. Perhaps this could be simplified by improving the `class RpnProcess`. Another way of improving could be to have some easily implementable standard scenarios that can be used in some often occurring situations.

2. Currently, the programmer is responsible for the process transformation functions. Perhaps this could be automated in some cases, for example by providing several standard functions, like the `Transform` class does.

3. At the moment, the network topology isn't analyzed by the network. If this would be done, then the network's transformation functions could be generated automatically, because in such a case, it is known in which order which process's transformation would be used in order to generate the correct transformation.

4. The model is now implemented in YAPI. It would be useful to be able to map the model or indeed a YAPI program on a multiprocessor platform that really executes the different processes on different processors.

5. While implementing the model, efficiency hasn't been a goal. Therefore, some functions could probably be implemented much more efficiently.

Despite these things, the implementation works and can be used to write and simulate RPNs. Several testing programs have been written and they proved that the system works. Two examples are used as illustrations in this report. There is an important limitation though: It is vital to the system that the number of tokens that is read and written are known a priori. When this is not the case, only 'nearly' synchronized reconfigurations are possible.

The theory has given a new perspective on the use of time in streaming networks and the implementation proofs that that perspective works. Hopefully, this will help developing new streaming data process networks in the future.

Bibliography

[1] F. Boussinot *Sémantique de réseaux parallèles : une approche du temps réel* Revue Technique Thompson-CSF, vol. 12, no. 3, September 1980

[2] Jack B. Dennis *First Version of a Data Flow Procedure Language*, MIT Project MAC Technical Memo. 61, May 1975

[3] Guillaume Doumenc and Frédéric Boussinot *La Programmation par Objets Réactifs*, Rapport EMP-CMA 15/92, 1992

[4] Tomas Henriksson, Jeffrey Kang and Pieter van der Wilf *Implementation of Dynamic Streaming Applications on Heterogeneous Multi-Processor Architectures* Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2005.

[5] Marc Geilen and Twan Basten *Reactive Process Networks* Proc. of the 4th ACM int. conf. on Embedded software, Pisa, Itali, 27-29 September 2004, pp 137-146, ACM Press, New York (NY), USA

[6] Gilles Kahn *The Semantics of a Simple Language For Parallel Programming* Information Processing 74: Proc of the IFIP Congress 74, Stockholm, Sweden, August 1974, pp 471-475, North-Holland Publishing Company, Amsterdam, The Netherlands

[7] Erwin De Kock and Gerben Essink *Y-chart Application Programmer's Interface*

[8] Stephen Neuendorffer and Edward Lee *Hierarchical Reconfiguration of Dataflow Models* Proc. 2nd ACM and IEEE Int. Conf. on Formal Methods and Models for Co-Design, 23-25 juni 2004, pp 179-188, IEEE, Los Alamitos (CA), USA

[9] Hristo Nikolov, Todor Stefanov and Ed Deprettere *Modeling and FPGA Implementation of Applications using Parameterized Process Networks with Non-Static Parameters* Proc. of the 13th Ann. IEEE Symp on FCCM'05, 2005

[10] Vincent Nollet *Run-Time Resource Management for Future MPSoC Platforms* 2008, TU Eindhoven, The Netherlands

[11] Martijn J. Rutten, Jos T.J. van Eijndhoven e.a. *Eclipse: Heterogeneous Multiprocessor Architecture for Flexible Media Processing* IEEE Parallel and Distributed Processing Symposium IPDPS 2002, 2002

[12] B.D. Theelen, M.C.W. Geilen e.a. *A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis* Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006.

[13] William Thies, Michal Karczmarek and Saman Amarasinghe *StreamIt: A Lanuage for Streaming Applications* Lecture Notes In Computer Science; Vol. 2304 Proceedings of the 11th International Conference on Compiler Construction, 2002