Behavioral Analysis of Real-Time Systems with Interdependent Tasks
Behavioral Analysis of Real-Time Systems with Interdependent Tasks

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 16 april 2008 om 16.00 uur

door

Mirela Alina Albu

geboren te Timisoara, Roemenië
Dit proefschrift is goedgekeurd door de promotor:

prof.dr. E.H.L. Aarts

Copromotoren:

dr. J.J. Lukkien

en

dr. P.D.V. van der Stok
Acknowledgements

During the last four years of my professional activity I have been supported by a number of people whom I would like to acknowledge in this section.

In the beginning I would like to express my gratitude to my supervisors Professor Emile Aarts, Dr. Johan Lukkien and Dr. Peter van der Stok. They are the ones who gave me the opportunity to continue my professional growth by participating in this project.

In the context of my PhD project Professor Emile Aarts acted as my first promoter. His openness and kindness in expressing his professional advice have counted to me a great deal. In truth, I can not remember ever leaving his office other than newly motivated in my work, and encouraged. For his support in key points of the project, his keen observations that increased the quality of the research, and career advice at the end, I am truly thankful.

I would also like to express my appreciation for the support I received from my University supervisor, Dr. Johan Lukkien. I would like to thank him for his genuine interest in my results expressed in the detailed technical discussions during the project. I am also grateful for the time and effort he took in reviewing my publications, I have been impressed to see the attention to detail he had in reviewing articles and this thesis as well. Overall, by his supervision, I would like to thank him very much for supporting my professional growth as a researcher, in the way I aspired at the beginning of this project.

Dr. Peter van der Stok acted as my company supervisor. In the first place I would like to thank him for organizing the context for my work in Philips Research. I am also grateful for his coaching and his thoroughness in reviewing my work throughout the project, Peter van der Stok has always been someone to go to with the last version of your work and his comments would make it even sharper.

I am also grateful for the review comments and support I have received during the past four years from the members of the users group of the STW/PROGRESS.

Along with my supervisors, there have also been a number of people who supported my work and I would like to remember them here. In that sense, regarding my colleagues in Philips Research, I would like to thank Liesbeth
Steffens, Laurentiu Papalau, Dietwieg Lowet, Clara Otero-Perez, Jeffrey Kang and Giel van Doren for the constructive discussions we had on this domain of research.

I address a special thanks to Dr. Hans van Gageldonk and Dr. Jos van Haaren for their support during the finalization phase of the thesis, while it has been reviewed and printed.

From the SAN group of the Eindhoven University of Technology, I wish to particularly credit Dr. Reinder Bril for his feedback, suggestions with respect to my papers and work in general, and above all for his kindness and willingness to help. To the SAN group in general I would like to address a big thanks for receiving me in their midst.

At the end I would like to thank my family without whom I would not have started this project in the first place.

With deep gratitude I would like to address my parents, Estera and Aurel Albu. Their many sacrifices and encouragement to pursue education in all seriousness since young ages allowed me to eventually reach this point. I will also never forget their last sacrifice to encourage me to pursue professional growth by continuing my studies abroad while knowing full well that I may never return. And while I know that they rejoice with me for each of my successes, I also know that the burden of that sacrifice is felt to this very day.

I would also like to thank my brother Tim Albu who was my first math tutor as a child and teenager. His goodness, patience and gift to make me like the things I learned, will stay with me forever. He was also the first person who strongly encouraged me and insisted that I should apply for this PhD position.

Among all those who supported me, I would like to mention Gerard Weffers whose genuine fatherly spirit, proud of my every result has always been very endearing to me during all these years.

In the end, I would like to address the person closest to me, Harold Weffers, my beloved husband. He is the one who for the past four years rejoiced together with me for each success and encouraged me through any disappointment. Thanks to his devoted care during our daily life I have been able to focus on my work in a degree otherwise impossible. His dear friendship and wisdom have always been the greatest support.

Eindhoven, December 2007

Alina Weffers-Albu
# Contents

1 **Introduction** ........................................................................... 1  
1.1 Media processing systems in home ......................... 1  
1.2 Problem statement ......................................................... 6  
1.3 Related work ................................................................. 9  
1.4 Thesis contribution ....................................................... 16  
1.5 Thesis outline .............................................................. 18  

2 **Trace Theory Concepts and Overall Approach** .......... 23  
2.1 Syntax ........................................................................... 23  
2.2 Semantics ...................................................................... 27  
2.3 States, invariants and channels .................................... 33  
2.4 The Streaming Pipeline ............................................. 35  
2.5 Approach summary ..................................................... 39  
2.6 Introducing timing constraints .................................... 40  
2.7 Summary ...................................................................... 48  

3 **A Linear Chain without Timing Constraints** ............... 51  
3.1 Characterization of the unique trace \( \rho \) ....................... 52  
3.2 Support for design practice ........................................... 55  
3.3 Summary ...................................................................... 61  

4 **Introducing Timing Constraints** ..................................... 63  
4.1 A linear chain with a time-driven component at the end . 65  
4.2 The interlaced standard ................................................. 80  
4.3 A linear chain where the first component is time-driven . 90  
4.4 A video surveillance system ........................................ 97  
4.5 Summary ..................................................................... 106  

5 **A study of components with deferred execution** ......... 109  
5.1 A linear chain where the first component is with deferred execution .................................................. 110  
5.2 Characterization of the unique trace \( \rho \) ................. 112  

vii
5.3 Adding a time-driven component at the end of the chain . . . . 122
5.4 A linear chain ending with a component with deferred execution 128
5.5 Summary ........................................ 132

6 Dealing with dependencies on the input stream content 135
6.1 A component with execution dependent on the input stream contents ................................. 136
6.2 A chain without timing constraints ......................... 140
6.3 A chain with timing constraints .............................. 148
6.4 A case study from practice - a video decoding chain .......... 155
6.5 Practical Applications .................................. 157
6.6 Summary ........................................ 157

7 A branching chain topology 159
7.1 A demultiplexer component .................................. 160
7.2 Assumptions ........................................ 163
7.3 QoS requirements ...................................... 165
7.4 System execution analysis .................................. 166
7.5 Practical applications .................................... 178
7.6 Summary ........................................ 179

8 Composition of media processing chains 181
8.1 Composition of chains consisting of only data-driven components ........................................ 182
8.2 Composition of chains with timing constraints ............... 182
8.3 Summary ........................................ 190

9 Conclusion 193

Bibliography 199

Publications 205

Symbol Index 207

Summary 213

Curriculum Vitae 215
1

Introduction

This thesis is concerned with the analysis of real-time systems with interdependent tasks such as media processing systems. Our aim is to characterize the behavior of these systems from which performance parameters such as start time and response time of individual tasks, chain end-to-end response time, number of context switches and resource utilization follow. Given the quality of service requirements of media processing systems, we investigate techniques for guaranteeing these system requirements as well.

We start this chapter by describing the domain of media processing systems in the home domain while underlining the close relation between satisfying the quality of service requirements of these systems and their real-time constraints. Next we formulate the problem statement relative to our goal and present how this problem has been approached in related work. Finally we present the thesis contribution and the outline of this book that includes a short description of each chapter.

1.1 Media processing systems in home

Media processing systems become increasingly pervasive in daily life. DVD recorders and players, video games, mobile phones that record and transmit short movies, PDAs, surveillance systems, radio stations on the World Wide
Web are only a few examples of such systems with which we are by now completely accustomed.

In general, media processing systems consist of a terminal side and a network side (Figure 1.1). Examples of terminal devices include DVD recorders and players, TVs, mobile phones, PDAs and PNDs. The communication media is implemented by means of wired or wireless interconnecting networks using for instance TCP/IP, UDP or RTP protocols [Tanenbaum, 2003].

![Figure 1.1. Interconnected media processing systems consisting of terminal devices and a network.](image)

### 1.1.1 Quality of Service and system real-time constraints

Market experience shows that while products offering completely new functionality are accepted in the beginning in spite of a lower level of quality in the service provided, as systems mature, robust versions are expected. For instance nowadays none of us would accept that while playing a movie at home, the DVD player would block at a frame and need resetting in order to be able to continue.

The services provided by the network concern the transmission of data between terminal devices while the terminal services regard for instance capturing, encoding, decoding, enhancement and rendering of the media. In general the term *service* refers to an encapsulated functionality provided by the system. The service that is experienced by the end user is the rendering of frames at a certain rate. However, in order for the system to be able to execute the rendering service, a few other services must execute and cooperate during their execution - in our discussion above, we mentioned the capturing and decoding services. All terminal services mentioned above are part of the *Application*
layer of the terminal system and they are built on top of other services executing at other levels in the system architecture (Figure 1.2):

- The **Middleware layer** that may implement quality of service and resource management policies
- The **Operating System layer** that takes care of task scheduling
- The **Network layer** which implements protocols that specify the way in which the data transmission is carried out. The Network layer [Tanenbaum, 2003] consists of further sub-layers which we do not detail here.
- The **Physical layer** which executes the system functionality both on the terminal and network side according to the policies implemented at the layers above.

![System architecture layers.](image)

But how does one assess a level in the quality of the service provided by these systems? And more fundamentally how is the quality of a service defined? In general, Quality of Service (QoS) as defined in the ITU-T Recommendation E.800 Geneva 1994 "... is the collective effect of service performances, which determine the degree of satisfaction for a user of a service".

To assess the level in the quality of the service provided by a system appropriate metrics need to be considered. For instance, for the network side of media processing systems the QoS levels are determined by the number of packets successfully delivered to the terminal, where success implies correctness of the transmitted data and transmission within the specified time. On the terminal side, one way to measure the QoS levels provided is the number of video/audio frames that are rendered at the appropriate time (the frame rate which should be for example in the case of video streams 25 frames per second in the case of the Phase Alternating Line (PAL) standard). Other QoS metrics used for video applications executing on the terminal side are the screen resolution, image size, color depth, bit rate and compression quality [Li & Nahrstedt, 1999], [Morros & Marqués, 1999], [Sabata, Chatterjee & Sydir, 1998].

The overall QoS level delivered by a system is influenced by the QoS level individually delivered by each service in the hierarchy of service layers presented in Figure 1.2. As a simple example consider a DVD player and a TV
which displays a video stream (a movie) stored on a DVD disk. The service experienced by the end user is the rendering of video frames at a specific rate. However, as we have seen above, the video rendering service is built on top of other services such as input data retrieval, decoding and video enhancement. The QoS levels delivered by each of these services individually (together with the QoS level of the video rendering service) influence the overall QoS level delivered by the system. Indeed if the decoding service delivers poor QoS levels then even if the video rendering service displays frames at the correct rate (which implies delivering a high QoS level), the displayed decoded frames will have artefacts which leads to an overall poor level of system QoS. The overall system QoS would also be affected if the situation were reversed meaning that the decoder service would decode frames at the highest level possible but the video renderer service would not display them at the correct rate. This shows that allocating large amounts of system resources to some services so they deliver the highest possible QoS levels does not induce an overall high level of system QoS if other services are "weaker links" delivering low levels of QoS. deliver

Returning to the robustness requirement mentioned at the beginning of this section, the criteria for robustness are defined with respect to the network part of these systems and the terminal side as well. In both cases robustness concerns meeting real-time constraints. In the context of the network, the real-time constraints come from the fact that media packets must be transmitted in time between terminals. The network real-time constraints are coupled to the terminal real-time constraints where the data must be received in time so that the audio/video frames are rendered at the appropriate rate in order to avoid audio/video artefacts. Note that the measure in which the real-time constraints are met is directly reflected in the value of the frame rate QoS metric at each execution moment. This implies that the degree in which the real-time constraints are met directly influences the QoS levels provided by the system as a whole (terminals and network).

1.1.2 Media processing systems on the terminal side

The results presented in this thesis have been produced in the context of the Quality of Service in in-home digital networks EES5653 PROGRESS project at Philips Research Laboratories in Eindhoven. The project represents part of on-going efforts towards implementing the concept of ambient intelligence [Aarts, Harwig & Schuurmans, 2001] in the context of a home environment. Such environments are characterized by multiple terminals cooperating in a distributed fashion.
The generic goal of the project was to provide guaranteed and optimised Quality of Service (QoS) for interconnected terminals, where the terminals are real-time embedded systems. The work presented in this thesis addresses QoS issues in the context of the terminal. The systems we study are built according to the Pipes and Filters architectural style [Buschmann & Et al., 1996] and its components are scheduled using fixed priority scheduling [Buttazzo, 2002], [Liu, 2000]. In this subsection we explain the type of systems that execute on the terminal side, and we explain the suitability of component based development combined with a Pipes and Filters architectural style and fixed priority scheduling for developing these systems.

Price erosion makes that high-end consumer products become main-stream in a short time. Therefore solutions are required that enable a short lead-time to introduce new features. To reduce this lead-time and costs associated with software development, designers needed to search for effective ways of constructing software for a family of products rather than for a single system. In that sense a promising approach was to build systems out of parameterized components, where as defined in [Maaskant, 2005], ”a software component is a unit of deployment that can be reused in multiple products (i.e. in multiple instances of the product family)”.

Furthermore, the type of processing performed by media processing systems on a terminal device implies applying a series of computations on the input media stream, where each computation is performed by a software component. As such, each component receives a fragment of the input data, modifies the data by means of some processing and passes on the result of the computation to another component. The last component renders the media either on a TV screen or at audio boxes. Because of this type of processing the Pipes and Filters architectural style comes as a natural choice in design and development. According to this architectural style, a media processing system on the terminal side can be viewed as a graph in which nodes represent software components and edges represent buffers. Each component corresponds to an operating system task, and the communication between tasks is buffered as shown in Figure1.3.

Our study concerns the situation where systems with characteristics as described above execute on a uni-processor platform. As we explain more detailed in the next section, the component tasks are interdependent, most of them are not periodic and only some of them have deadlines. The variety in the tasks behaviour induces a significantly complex overall system behavior. To ensure the control on the way the system resources are consumed, scheduling is needed. Fixed priority scheduling is preferred in industrial practice over other scheduling policies due to its straight forward way of use. Another reason
is that assigning static priorities to the system tasks makes the system execution more predictable as opposed to dynamic methods, like for example, the Earliest Deadline First (EDF) scheduling.

Nevertheless, as we will see in the next section, predicting the execution of the system remains a challenging task. We explain this challenge in the problem statement presented next, and we describe our approach to this problem in the thesis contribution section.

1.2 Problem statement

The work presented in this thesis focuses on QoS issues for media processing systems in the context of the terminal. The problem we address in particular is how to build media processing systems that satisfy QoS requirements while using a minimum of resources. This leads to further questions about constructing and modifying a media streaming system executing on the terminal side such as:

- How much resources does the system consume?
- Given a certain amount of resources, will the system meet its timing and QoS requirements?
- What is the minimum amount of needed resources such that timing and QoS requirements are met?
- What can one do to reduce the resource needs of the system?
- What can be done to improve the extent to which the system meets timing requirements?
1.2 Problem statement

The questions above could easily be answered if the overall behavior of the system could be predicted and controlled at system design time. That way, performance parameters that characterize the timing properties and resource consumption of the system could be calculated and optimized already at system design time. For media processing systems executing on the terminal side these parameters are

- Response times of tasks and individual chains of components,
- Minimum necessary and sufficient memory buffer capacities to avoid deadlock and to meet timing constraints,
- CPU utilization,
- Number of context switches and the context of their occurrence during the execution, which give an indication about the overhead introduced.

By calculating at design time the values of these parameters, one could predict whether the overall system satisfies its timing constraints and hence its QoS requirements. Moreover, by learning how to control the behavior of the system at design time, we could build optimized systems that are guaranteed to satisfy their timing and QoS requirements.

As it turns out, predicting and controlling the overall behavior of the system is quite challenging. We dedicate the rest of the section to explain these challenges.

Low predictability due to scarcity of resources leading to resource sharing

The first difficulty comes from the fact that the high production volume of terminal devices sets severe requirements on the product cost, leading to resource-constrained devices. The scarcity of resources induces sharing between the component parts of a system. For this reason it is difficult to predict which component holds the system resources when, for how long.

Low predictability and control due to the complex set of factors determining the overall system execution

A second challenge comes from the fact that the combined execution and performance of the components is determined by a multitude of factors. The components that constitute a media processing system on the terminal side belong to different component types, which induce different component behaviors. These component types are:

- Data-driven components
- Time-driven components
- Components with deferred execution
Components with *execution dependent on the input stream contents*

*Demultiplexer* components

*Mixer* components

The *data-driven* components have the least complex behaviour. Their execution is determined by the availability of the necessary input and the priority of the associated component tasks. These components are usually used to improve the quality of decoded frames such as the sharpness enhancement component.

The *time-driven* components have a periodic behaviour. Their execution is determined by the availability of the necessary input, the priority of the associated component tasks and the component tasks period. Examples of such components include the video digitizer, video renderer and audio renderer.

Compared to the *data-driven* components, the behaviour of components with *deferred execution* is also influenced by the duration of the deferral types. Examples of such components are the file reader and file writer components that respectively retrieve the input stream from a storage facility (i.e. DVD disk, hard disk) and store the stream on a hard disk.

Components with *execution dependent on the input stream contents* are in general video and audio decoders or encoders. Their behaviour is highly variable and dependent on the input stream contents due to the fact that input frames have usually different sizes and depending on their type, require different computation times to be processed [Baiceanu, Cowan, McNamee, Pu & Walpole, 1996], [Lan, Chen & Zhong, 2001], [Peng, 2001], [Zhong, Chen & Lan, 2002]. For instance in the MPEG2 standard, I frames are generally larger than P or B frames. Hence an I frame will in general be stored over a larger number of input packets (of fixed size) compared to the B or P frames. This implies that in the case of a video decoder, the number of input packets needed to start processing is variable, depending on the size of the next encoded frame relative to the fixed size buffer packets. Also with respect to the computation times needed to decode an input frame, I frames usually need less time to be decoded than B or P frames. Again, for these components as well, their execution is determined by the availability of the necessary input and the priority of the associated component tasks.

*Demultiplexer* components take as input program or transport streams, split the video data from the audio data and pass it on the corresponding video or audio decoding chain.

Finally, the *Mixer* components are the reverse of the *Demultiplexers*. They take as input multiple streams and create a mixed output to be rendered on the TV screen. Examples of systems using the *Mixer* component provide the
1.3 Related work

We present the related work from different perspectives according to which we relate our contribution.

**Real-time theory for interdependent tasks**

Classic real-time theory mainly focuses on analyzing the execution of independent periodic tasks. In [Buttazzo, 2002] the subject of tasks dependency is mentioned when presenting tasks with precedence constraints, and mutual exclusive executions.

In the extended literature, several attempts have been made to analyze message passing, streaming systems. Closely related work [Groba, Alonso, Rodrígues & García-Valls, 2002] considers also an execution model for video streaming chains inspired by TSSA. The article [Groba, Alonso, Rodrígues & García-Valls, 2002] presents an analysis method allowing the calculation of the worst-case response time of multiple video streaming chains based on the canonical form of the chains. The assumptions adopted are that tasks have fixed execution times, tasks are allowed to have equal priorities and the overhead introduced by context switches is ignored. Their approach is based on the response time analysis for tasks with deadlines beyond periods [González-Harbour, Klein & Lehoczky, 1991].

Klein et al. [Klein, Ralya & Et al., 1993] apply fixed-priority response time analysis to message-passing systems. The system is modeled in terms of events and event responses. Message handlers create new events when outgoing messages are sent at a different rate than incoming messages. Tasks are modeled as shared resources. The processing of a message by a task is modeled as an
atomic action on the shared resource. This leads to the response-time analysis of a set of independent event responses with atomic access to shared resources.

Goddard [Goddard, 1997] studies the real-time properties of PGM data flow graphs [Bhattacharyya, Murthy & Lee, 1996], which closely resemble our media processing graphs. Given a periodic input and the data flow attributes of the graph, exact node execution rates are determined for all nodes. This is very similar to the approach presented in [Klein, Ralya & Et al., 1993]. The periodic tasks corresponding to each node are then scheduled using a preemptive EDF algorithm. For this implementation of the graph, the author shows how to bound the response time of the graph and the buffer requirements. Both approaches consider complete task sets scheduled by a single scheduling algorithm, and are limited to task sets with deadlines equal to the period, i.e. without self-interference.

QoS improving techniques for media processing systems on the terminal

*Techniques for improving QoS at system level*

*Resource reservation* [Mercer, Savage & Tokuda, 1994], [Otero-Pérez, Rutten, Steffens & Van Eijndhoven, 2005] is the process of allocating and guaranteeing (enforcing the allocation) amounts of resources to an application. Given the direct correlation between the level of QoS provided by an application and the resources needed to provide that QoS level, resource reservation is a straightforward technique that ensures the provision of QoS by enforcing the availability of resources to the application [Lee, Lehoczky, Rajkumar & Siewiorek, 1999], [Audsley, Burns, Richardson & Wellings, 1993], [Sprunt, Sha & Lehoczky, 1989].

Resource reservation is based on the concept of *budget* that defines the amount of resources available to an application per unit of time [Rajkumar, Juvva, Molano & Oikawa, 1998], [Caccamo, Buttazzo & Sha, 2000], [Lipari & Baruah, 2000]. Budgets are part of the means to solve conflicts when multiple applications execute on the same platform and hence must share its resources. However, especially for media processing applications, budgets do not provide the entire answer to resource sharing. That is because the processing load of each of the application varies depending on the input stream contents [Baiceanu, Cowan, McNamee, Pu & Walpole, 1996]. For this reason in the case of these applications budgets are used to guarantee the average amount of resources need by an application, while the variations around that average are handled by the application itself. In other words applications must be able to handle situations in which the needed resources are less than the available budget.
Application adaptation is an approach in which applications adapt their behavior and their requirements in terms of QoS and (implicit) resources to the resource availability at hand. This technique can be implemented by allowing tasks with asynchronous communication to work ahead in order to balance the load on the processor [Wüst, Steffens, Bril & Verhaegh, 2004], [Sha, Lehoczky & Rajkumar, 1986] or by using scalable video algorithms in which certain tasks can operate in different modes corresponding to different levels of quality in the output and different resources needs [Hentschel, Bril & Chen, 2002], [Hentschel, Bril, Chen, Braspenning & Lan, 2002], [Wüst, Steffens, Verhaegh & Et al., 2005], [Lafruit, Nachtergale, Denolf & Bormans, 2000] describe methods to regulate varying computational load for high-quality video decoding and for 3D decoding and rendering, respectively, assuming synchronous processing.

A combination of resource reservation and application adaptation was used in the QoS-RM project at Philips Research Laboratories Eindhoven. The approach taken was to divide the overall system in sub-systems that can be allocated individual resource budgets. Such sub-systems have been called Resource Consuming Entities (RCE). Application adaptation is implemented by allowing RCEs to execute in different modes providing different levels of quality and thus requiring different amounts of resources. A mode provides a number of corresponding quality of service levels. Resource reservation is implemented in that for each RCE executing in a particular mode within which a particular quality level was selected, a particular amount of resources is allocated and guaranteed (budget). The module that monitors the allocation of resources in the terminal is called Budget Manager. As such, for any request of resources in the terminal the Budget Manager checks if the requested amount is available in the system (admission control) and if that is the case then it allocates and enforces the requested amount for the RCE for which the request was made (resource reservation). If the amount of resources requested is not available in the system, the Budget Manager denies the request and as a consequence the RCE will not be able to run in that mode. This means that the RCE will attempt to execute in a different mode which requires less resources.

Another example of combining resource reservation and application adaptation is presented in [Bril, 2004]. In this approach resource reservation is based on conditional guaranteed budgets (CGB). Conditional guaranteed budgets can be allocated in two ways: a weak CGB or a strong CGB. A weak CGB is allocated based on the surplus time of a CGB provider, and can only be weakly guaranteed, even when that surplus time is available consistently. Strong CGBs are based on the assumption that a structural load increase is anticipated timely based on knowledge about the input stream contents.
The combination of application adaptation and resource reservation has also been described in [Foster, Roy & Sander, 2000] and [Hamann, Löser, Reuther, Schönberg & Wolter, 2001].

In [Pastrnak, De With, Ciordas & Van Meerbergen et al., 2006], [Pastrnak, De With & Van Meerbergen, 2006] and [Pastrnak, 2008] the authors describe a method that combines reservations with a best-effort run-time adaptation of the computation in the case of media processing systems executing on a multi-processor platform. The study focuses on presenting the benefit of adding best-effort computing services for the communication within an MP-NOC to improve the efficiency in cases where resources remain unused due to the fluctuating resource needs of some tasks. More specifically, given the fact that in the case of MPEG-4 decoding the amount of objects is variable implying that the decoding process is highly variable in resources usage, the execution is ensured to have guarantees on decoding at the lowest quality. The higher quality levels are provided by adding best-effort tasks to the reservation-based processing at the lowest quality.

The concept of combining guaranteed services with best-effort services is visited also in [Rijpkema, Goosens & Et al., 2003] and [Goosens, Van Meerbergen, Peeters & Wielage, 2002]. In [Rijpkema, Goosens & Et al., 2003] the authors present a router-based NOC architecture consisting of two parts: the guaranteed-throughput (GT) router and the best-effort (BE) router. The guarantees are never affected by the best-effort traffic, while the BE traffic uses all the bandwidth left over by the GT traffic.

Finally, overprovisioning is a technique that provides an easy solution for QoS by providing an abundance of resources such that the service will always be able to provide the highest level of quality. Although straightforward to implement, obviously the disadvantage of this technique is that it is inefficient and expensive especially in the case of media processing systems that experience highly variable resource needs. Overprovisioning in this case implies that large amounts of resources made available for the worst case scenario in terms of resource requirements, are needed and used completely only some of the time.

Techniques for improving QoS at algorithm level

As we have seen in the previous sections, satisfying QoS requirements implies satisfying the real-time constraints of the system which in the case of media processing means that the video and audio renderer component tasks must render the video/audio information at the appropriate time. However, as we have seen above, sometimes an application will not have enough resources to be able to (for instance) decode the input stream fast enough such that each frame
is rendered in time. One approach to this situation is that the decoder component estimates the decoding and presentation time of the frame in processing. If these estimated times are too late then the decoder component "drops" the current frame and continues processing with the next frame in the input buffer. This approach has been presented in [Isović & Fohler, 2004]. The approach presented above allows only the rendering of those frames that can be decoded in time. The advantage of this method is that the frames that are rendered, are decoded at the highest level of quality possible. The disadvantage is that frames that could have been decoded if the application had had just "a little more" resources will not be rendered at all.

In contrast to the afore mentioned approach, some of the techniques that address QoS issues at the level of algorithms provide solutions to the "all or nothing" situation presented above. For instance a video application may temporarily drop the decoding quality level, to alleviate overload problems [Wubben & Hentschel, 2003]. The algorithm implementing the decoder mentioned before belongs to a special class called scalable video algorithms (SVA).

In general an SVA consists of an algorithm that handles the media processing and a quality control block [Hentschel, Braspennin & Gabrani, 2001]. The algorithm incorporates a number of specific functions, some of which are implemented to be scalable. That means that depending on the available resources, each of these functions can execute in a different mode (at a different quality level) that ultimately determines the quality level of the output. The overall quality of the output depends on the appropriate combination of the quality levels of these functions. The optimal quality-resource combinations correspond to optimal points obtained using a Pareto curves analysis. Given the amount of resources available at a certain moment, the quality control block determines which are the most appropriate modes of execution for each of the specific functions such that the overall quality of the output is maximized.

In essence SVAs allow making trade-offs between resource needs, and output quality while guaranteeing that the real-time constraints of the media processing system are satisfied. That means that the value of the frame rate QoS metric is always correct. The authors of [Lan, 2001] and [Peng, 2000b] give an insight into the types of trade-offs they focused on when presenting their complexity-scalable MPEG2 decoder with graceful degradation. Examples of techniques they implemented for scalable MPEG2 decoding were:

- **Graceful degradation** which deals with the trade-off between the compute resource (complexity) and the output quality [Peng, 2000a], [Zhong, 2000].
• **Embedded resizing** dealing with the trade-off between the compute resource (memory, memory bandwidth and complexity) and output image size [Zhong, 2000], [Zhong, Peng & Van Zon, 1999].

Solutions describing computational complexity scalable video processing are described in [Peng, 2001], [Mietens, De With & Hentschel, 2004a], [Mietens, De With & Hentschel, 2004b], [Mietens, 2004] [Mietens, De With & Hentschel, 2003].

In [Wüst, Steffens, Verhaegh & Et al., 2005] the authors explain an approach that allows close-to-average-case resource allocation to a single video processing task, based on asynchronous, scalable processing, and QoS adaptation. The QoS adaptation balances different QoS parameters that can be tuned, based on user-perception experiments: picture quality, deadline misses, and quality changes.

Other examples from literature that address the SVA domain include [Lan, Chen & Zhong, 2001] who also described a scalable MPEG decoder which estimates the resource needs before decoding a frame and scales the decoding such that it will not exceed the target computation constraint. In contrast to the approach in [Wüst, 2006], they only optimize the output quality of individual frames and not the overall perceived quality over a sequence of frames.

In [Jarnikov, Van der Stok & Wüst, 2004] the authors tackle an additional problem, that of an input signal with fluctuating quality. In their approach each encoded frame consists of a **base layer** and a number of **enhancement layers**. Depending on the available resources, the video decoder may decide to decode only the base layer, or to follow up with a number of additional enhancement layers. Decoding only the base layer produces poor output quality results, nevertheless the timing constraint of the system is respected and the frame is not dropped. Decoding additional enhancement layers on top of the base layer improves the output picture quality. Alternative scalable video coding solutions are described in [7, 8, 9, 10D]

**Techniques for improving user perceived QoS**

The techniques surveyed so far, aim to improve the QoS from an objective point of view determined by specific values attached to QoS metrics. Unfortunately, experience shows that sometimes by attempting to improve the values of the QoS metrics, the quality of the (human) user experience while using the service is not changed, or on the contrary is diminished. The quality of the human experience while using a service is named in literature **perceived quality**, or user perceived QoS.

For instance, in [Jarnikov, Van der Stok & Wüst, 2004], [Jarnikov, 2007] we have seen that the quality of a frame depends on the number of decoded lay-
1.3 Related work

ers corresponding to the frame. The more layers decoded, the better the quality of the frame. However experience shows that the perceived quality can be affected negatively in the case of viewing a sequence of frames (a movie) with fluctuating numbers of layers belonging to the constituent frames. Although in some cases while decoding the movie sequence there would be enough resources to decode frames from time to time at a higher quality, in doing so the user will experience a changing in the quality of the image which is perceived to be more disturbing than if the entire sequence would be decoded at a lower quality.

In that sense we consider very relevant the work presented in [Zink, Künzel, Schmitt & Steinmetz, 2003] where the authors present an assessment of video quality relative to the influence of the amplitude and frequency of layer variations. The amplitude is defined as the height of a layer variation while the frequency determines the number of layers variations. A segment is an equal-sized time unit per layer. We present below a few of the techniques for achieving higher levels of user perceived QoS produced by their studies:

• Stepwise decrease of layer encoded video amplitude: a stepwise decrease is rated better than one single but higher decrease.

• Low layer encoded video frequency: decode less but a constant amount of layers.

• Closing the gap: if in position to choose, closing a gap on a lower level results in a better quality than closing a gap on a higher level.

• Decrease vs. increase: starting with a higher amount of layers, decreasing the amount of layers, and increasing the amount of layers in the end again seems to provide a better perceivable quality than starting with a low amount of layers, increasing this amount of layers, and going back to a low amount of layers at the end of the sequence. This might be caused by the fact that test candidates are more concentrated in the beginning and the end of the sequence.

• More quality at the end: increasing the amount of layers in the end leads to a higher perceived quality.

Also relevant for determining and measuring user perceived QoS are the contributions in [Nelakuditi, Harinath, Kusmierk & Z.-L.Zhang, 2000] and [Rejaie, Handley & Estrin, 1999] where the authors research user perceived QoS metrics. In [Nelakuditi, Harinath, Kusmierk & Z.-L.Zhang, 2000], Nelakuditi et al. state that a good metric should capture the amount of detail per frame as well as its uniformity across frames. Their quality metric is based on the principle of giving a higher weight to lower layers and to longer
runs of continuous frames in a layer. The quality metric presented by the work of Rejaie et al. [Rejaie, Handley & Estrin, 1999] incorporates as parameters completeness and continuity. Completeness of a layer is defined as the ratio of the layer size transmitted to a terminal for decoding compared to its original (complete) size. Continuity is the metric that covers the gaps in a layer. It is defined as the average number of segments between two consecutive layer gaps.

**Behaviour simulation approaches for terminal media processing systems**

In [Bondarev, Pastrnak, De With & Chaudron, 2004] the authors propose a scenario simulation approach for predicting the timing and resource usage of component-based media processing systems at design time. The authors validate the technique on a case study that regards the development of an MPEG-4 video application. The proposed approach is based on three concepts:

- models for the system component’s behaviour and resource usage
- execution scenarios of the complete system, in which the resources are potentially overloaded
- simulation of these scenarios, resulting in timing behaviour of the designed system.

In [Bondarev, Muskens & De With et al., 2004] the simulation approach includes the possibility to handle mutual exclusion, combinations of aperiodic and periodic tasks and synchronization constraints. The simulator provides data about dynamic resource consumption and real-time properties like response time, blocking time and number of missed deadlines per task.

### 1.4 Thesis contribution

We compare our contribution along the three directions presented in the previous section.

**Real-time theory for interdependent tasks**

Within this direction, each of the approaches presented provided valuable insights, but none of them helps engineers to reason in the detail we need about system behavior and associated resource needs.

Our research aims at providing an underlying theory that helps engineers to reason rigorously about system behavior and associated resource needs. It starts from the experimental observation that a media processing chain, assumes a repetitive behavior, the *stable phase*, after a finite *initial phase*. Starting from this observation we are building a theoretical model for the execution
of streaming graphs in media processing systems. Our general strategy is to analyze streaming systems in an incremental manner starting from a simple theoretical case, to realistic streaming chains that include branching and complex types of components mentioned in section 1.2.

Our approach allows us to calculate the execution order of the components in a chain, expressed as a trace of actions [Hoare, 1985] taken by each component at system design time. We formally prove that the behavior of the chain can be expressed as a unique trace, which assumes a repetitive pattern after a finite prefix. The trace is completely determined by

- the individual traces of the components determined by their type,
- the timing behaviour of components,
- the topology of the system,
- the capacities of the communication buffers,
- static priorities of the components.

The unique trace of actions proves an excellent starting point for further analysis. The initial phase can be calculated and optimized. Simple additive formulas for the start times and response times of the individual tasks and the complete chain are immediately available. The number of context switches, and the position of the context switches in the component traces, which is an indicator for their overhead cost, can be extracted from the trace. Also given the individual traces of the components and the channel constraints (due to the asynchronous communication), we calculate the necessary and sufficient capacities for each buffer in the chain such that deadlock will not occur and overprovision in terms of the processing power is avoided for systems confronted with highly variable computational needs. Hence the approach also allows the calculation and optimization of the capacities of the queues between components.

The repetitive nature of the chain is an important property that also makes reasoning about composition of chains much easier. Designers need only to reason in terms of patterns of execution at the level of the chain instead of reasoning about the individual behaviors of components within the whole system. This approach also makes systems open in the sense that the effect of inserting (or withdrawing) components from a chain can be rigorously predicted and controlled.

**QoS improving techniques for media processing systems on the terminal**

Our work describes techniques for guaranteeing QoS requirements at system
(RCE) level. The QoS metric that we focus on is the frame rate. The techniques we propose and formally prove are to be applied at design time and are based on:

- the relation between the sum of the computation times of one loop iteration of the components and the required frame rate,
- a specific priority assignment to the component tasks,
- a minimum necessary and sufficient of buffer capacity in a specific buffer as a trade-off for less processing power such that overprovisioning is avoided in the case of systems requiring highly variable processing load.

**Behaviour simulation approaches for terminal media processing systems**

Finally, when comparing our contribution with the related work described in the previous section along this direction, we notice a few aspects.

The work is related in that we address the same class of systems - media processing systems designed using a pipes and filters architectural style. However while the authors referred to in the related work section use a simulation based approach, our approach is analytic. We model the system we study, and formally prove properties about its behaviour. These properties are further used to prove optimization techniques regarding the system resources and timing properties as well.

An important point is that in the simulation based approach the validation of results is given on numerous specific scenarios. In our approach the analysis reveals an inherent behavioral property of this class of systems, designed according to the pipes and filters architectural style and scheduled according to fixed priority scheduling. This is very relevant because it implies that the results we present hold for any input streams used and for any scenarios that conform to the class of systems we address.

### 1.5 Thesis outline

In Chapter 2 we establish a number of basic concepts used in our theory, and we formally introduce the systems we study. Our aim is to analyze the system behavior by focusing on the corresponding trace set that contains traces recording all potential executions of the system and the actual one. In this chapter we present our approach to identifying the trace that records the actual execution of the system. The conclusion of the chapter is that both in pipelined systems with and without timing constraints, there exists a unique trace \( \rho \) that specifies the system execution.
In Chapter 3 we analyze the behavior of a pipelined system without timing constraints, where all components in the system are data-driven. The system we address is a linear media processing chain executing in a cooperative environment. We analyze the system behavior by studying the unique trace $\rho$ that records its actual execution. We formally prove that the trace becomes repetitive (the stable phase) after a finite prefix (the initial phase) and we show that this trace can be calculated at design time. This approach allows the calculation and optimization of the capacities of the queues between components, of the initial phase, of the number of context switches, and of the response time of individual components and the entire chain.

In Chapter 4 we analyze the behavior of linear chains that contain data-driven and time-driven components. In contrast to the system studied in Chapter 3, the systems we analyze in this chapter have timing constraints, and QoS requirements. We prove that a time-driven component in a chain where all the other components are data-driven, has the same influence on the overall execution of a chain as a data-driven component with minimum priority has on a chain composed of only data-driven components. This reduces the analysis of this time-driven system to be identical to that of the data-driven system in Chapter 3. In the case where a system contains two time-driven components, we show how the system can be designed such that all components develop a dependency on only one of the time-driven components. One other important result of this chapter refers to CPU overload situations in which the time-driven component at the end of the chain misses its deadline for a number of periods. In these cases we show how to design the system such that there always exists an infinite suffix of the trace $\rho$ during which the chain satisfies the QoS requirements. The results of this analysis are relevant because they show a cheap solution at design time of systems that guarantees meeting QoS requirements for an infinite suffix of the system trace. The solution is suitable for systems that experience high variations in computation times of tasks and it concerns trading off small additional amounts of memory in a specific buffer for much lower processing power.

In Chapter 5 we introduce a new type of components called components with deferred execution. The analysis of this chapter shows what is the influence of adding a component with deferred execution to the systems previously studied in Chapter 3 and Chapter 4. In all analyzed cases we prove the repetitive nature of the system execution and we analyze the influence of the component with deferred execution on the overall execution of the system. Practical applications regard again techniques for meeting the QoS requirement, optimization of memory, number of context switches and response time. A distinguishing issue tackled here is the optimization of CPU utilization by
eliminating the potential idle times occurring during the deferral times of the component with deferred execution.

In Chapter 6 we study the influence of the input stream contents on the overall execution of a media processing system. This influence comes as a result of the fact that the behaviour of some components in the system changes depending on the input stream contents (components with execution dependent on the input stream contents). Building towards realistic systems, we adopt an incremental approach starting from a system without timing constraints to systems with timing constraints. The aim is to study the influence of the component with execution dependent on the input stream content on the overall systems behaviour. In both cases we show that the traces that record the execution of the systems we analyzed adopt a repetitive pattern dependent on the contents of the input stream. The two patterns correspond to the two execution scenarios of the component dependent on contents of the input stream. For this new pattern of execution we address again practical applications concerning QoS, optimizing system resources and timing properties.

In Chapter 7 we study the execution of a system consisting of a linear sub-chain connected to two other linear sub-chains with timing constraints. The main difference between the system studied here and those studied in all previous chapters is the system topology: in the previous chapters we have studied linear chains while in Chapter 7 we tackle the analysis of a system with branched topology. Aside of that, in the present case we also introduce a new type of component, the demultiplexer. In the system we study here, the sub-chain that receives input from the environment, contains a component with deferred execution and a demultiplexer component that provides input to the other two sub-chains that follow in the graph. Each of the other two sub-chains consists of a component whose execution depends on the content of the input stream, a number of data-driven components and end with a time-driven component. As a first step, given the assumptions considered, we characterize the execution of each sub-chain within the overall execution of the system, and subsequently we characterize the interleaving of these two executions while pointing out what are the situations in which the QoS requirements are satisfied. In characterizing the individual execution of each sub-chain within the overall system execution we use a similar approach as presented in Chapter 6. We explain that the actual interleaving between the executions of the two composing sub-chains is determined by the ratio between the periods of the two time-driven components at the end of the two sub-chains, the contents of the input stream which influences the computation times of the trace actions, the duration of the deferral times of the first component and the priority assignment of the components. Practical applications concerning QoS, optimizing
system resources and timing properties are addressed again at the end of the chapter.

In Chapter 8 we analyze the behaviour of systems composed of two independent linear chains as opposed to Chapter 7 where we studied the composition of two dependent chains. We tackled two types of independent composition: where none of the chains have timing constraints and the situation where both chains have timing constraints. In the first case both chains are composed of only data-driven components. We show that composing these chains is not advisable because after a finite prefix one of the chains becomes starved. In the second case the first chain corresponds to a video decoding chain and the second chain to a surveillance application that saves on the hard-disk the images captured by the first component. The challenge in this case is to find solutions for designing the composition of the chains such that both chains satisfy their QoS requirements. We show that certain priority assignments imply supplementing the buffer capacities in the chains which is costly. We propose and detail a cheaper solution in which the buffers do not need to be larger than one position each. Our solution to satisfying the QoS requirement is to impose a specific priority assignment to the components and to control the phasing between the executions of the two systems. We also show how to design a system such that the necessary condition for the phasing is satisfied.

Finally in Chapter 9 we present the conclusions of the studies presented in this thesis.
In this chapter we establish a number of basic concepts used in our theory, and we formally introduce the systems we study. The systems we address in this work are composed of a finite number of software components. Each component is specified by means of a program text (section 2.1). We use an imperative programming language much like C or Pascal for that purpose. The semantics of a program is given by a set of traces, each trace specifying a potential execution of a machine according to the program (section 2.2).

Our aim is to analyze the system behavior by focusing on the corresponding trace set. The trace set contains traces recording all potential executions of the system and the actual one. In this chapter we present our approach to identifying the trace that records the actual execution of the system. Sections 2.4 and 2.5 detail this approach for systems without timing constraints, while section 2.6 analyzes systems with timing requirements.

2.1 Syntax

We use a simple intuitive syntax for the program text of the components using repetition (‘while’), selection (‘if’), sequential composition (‘;’) and basic
\begin{align*}
\textit{PROGRAM} & \rightarrow '{\{' \textit{SC} '{'} \}}'. \\
\textit{SC} & \rightarrow \textit{SC} '{'} \textit{SC} \\
& \mid \textit{S} \\
\textit{S} & \rightarrow \textit{B} \\
& \mid \textit{CS} \\
\textit{B} & \rightarrow \textit{skip} \\
& \mid \textit{VAR} '{'} := '{' \textit{EXPR} '{'}' \\
& \mid \textit{receive} '{' \textit{Q}, \textit{VAR}, \textit{NUMBER} '{'}' \\
& \mid \textit{send} '{' \textit{Q}, \textit{VAR}, \textit{NUMBER} '{'}' \\
& \mid \textit{process}_\textit{fact} '{' \textit{VAR LIST} '{'}' \\
\textit{G} & \rightarrow \textit{EXPR} \\
\textit{CS} & \rightarrow \textit{if} '{' \textit{G} '{'}' \textit{then} '{' \textit{SC} '{'}' \textit{else} '{' \textit{SC} '{'}' \\
& \mid \textit{while} '{' \textit{G} '{'}' \textit{do} '{' \textit{SC} '{'}' \\
\end{align*}

Figure 2.1. Generative grammar specifying rules according to which component programs are constructed.

statements for communication and computation. The set of basic statements and tests (guards) of program \( C \) specifying a software component is called its alphabet \( A(C) \). Alphabets of different components are disjoint.

Consider the following grammar (Figure 2.1) describing the syntax of component programs that we study. \( \textit{SC} \) denotes \textit{sequential composition}, \( \textit{S} \) stands for \textit{statement}, \( \textit{B} \) stands for \textit{basic statement}, \( \textit{G} \) for guards, \( \textit{CS} \) for \textit{compound statement}, \( \textit{Q} \) for \textit{queue}, \( \textit{VAR} \) for \textit{variable}, \( \textit{EXPR} \) for \textit{expression}, and \( \textit{NUMBER} \) for a numeric constant. \( \textit{VAR} \) and \( \textit{EXPR} \) are non-terminal symbols that can easily be defined by additional grammar rules which will not be included here. Statements \textit{receive}(\textit{Q},\textit{VAR},\textit{NUMBER}) and \textit{send}(\textit{Q},\textit{VAR},\textit{NUMBER}) represent the retrieval and respectively sending of data via queue \( \textit{Q} \). Argument \( \textit{VAR} \) stands in this case for an array and \( \textit{NUMBER} \) the number of items to retrieve.
or send. Statement \( \text{process}_{\text{fct}}(\text{VAR LIST}) \) represents the processing body of a component. The list of arguments represents the input variables and the variable in which the result of the processing is stored. A simple example of a component program derived from this grammar is:

\[
\text{C} : \{ x := a + b; \; y := b - c; \; z := c * a; \; p := d / b \}
\]

This program consists of four statements, each of them specifying a variable being assigned the result of an expression. Note that we used \( \text{C} : \) as label indicating the beginning of the program associated with component \( \text{C} \).

Another example of a program specifies an infinite repetition of two assignments as presented below:

\[
\text{C} : \{ \text{while (true) do } \{ \; x := a + b; \; y := b - a \; } \}
\]

Having introduced the basic concepts about component programs we continue with showing the types of components that are part of the systems we study. We focus on systems consisting of a collection of communicating components connected in a pipelined fashion (conform the Pipes and Filters architecture style). An instance of this architecture style, the TriMedia Streaming Software Architecture (TSSA) provides a framework for the development of real-time media streaming systems executing on a single TriMedia processor. A media processing system is described as a graph in which the nodes are software components that process data, and the edges are channels that transport the data stream in packets from one component to the next. The channels are implemented by finite queues (buffers). In the remainder of this thesis we will use interchangeably the terms channel and queue referring to the same notion.

A simple example of such a chain is presented in Figure 2.2.

![Figure 2.2. Chain of components.](image)

Every connection between two components is implemented by two queues. One queue (forward queue) carries full packets containing the data to be sent from one component to the next, while the second queue (backward queue) returns empty packets to the sender component to recycle packet memory. The empty packets are returned to signal that the data has been received properly and that the associated memory may be reused.

We denote with \( \text{Cap}(q) \) the capacity of queue \( q \). We also denote with \( L(q) \) the length of \( q \), where the length expresses the number of elements currently stored in \( q \). The capacity of \( f_{qi} \) is equal to the capacity of \( b_{qi} \). The system executes in
a cooperative environment meaning that the environment will always provide input and always accept output. That means that blocking on the queues \(fq_0, bq_0, fq_N, bq_N\), is not possible. The initial situation of the chain is that all forward queues (except \(fq_0\)) are empty and that all backward queues (except \(bq_0\)) are filled to their full capacity. This is expressed as

\[
L_0(fqi) = 0 \land L_0(bqi) = \text{Cap}(bqi) \quad \forall i, 0 < i \leq N.
\]

An example for the behavior of \(C_i\), \(1 \leq i \leq N\), is the following (Figure 2.3): the component receives 1 full packet (FP) from the input forward queue \(fqi_{i-1}\) \(\textcircled{1}\), then receives 1 empty packet (EP) from the input backward queue \(bqi\) \(\textcircled{2}\), performs the processing \(\textcircled{3}\), recycles the input packet from \(fqi_{i-1}\) by sending it in the output backward queue \(bqi_{i-1}\) \(\textcircled{4}\) and finally, the result of processing is sent in the output forward queue \(fqi_i\) \(\textcircled{5}\). Relevant to note here is that different component behaviors induce a different behavior of the overall system.

Figure 2.4 shows the program describing this execution of a component \(C_i\), syntax derived from the grammar we presented at the beginning of this section:

Finally we need to extend the grammar to be able to generate programs implementing systems composed of a number of components as seen in Figure 2.2. The component programs execute concurrently while in themselves do not have concurrency. The semantics attached to this program composition is similar to the one presented in [Hoare, 1985], [Van de Snepscheut, 1993], [Lukkien, 1991]. Informally, execution of \(P_1 || P_2\) means that the basic statements and tests of \(P_1\) and \(P_2\) are interleaved in a non-deterministic way. A more precise definition regarding the semantics of the parallel composition of two programs will be provided in the next section. We present the extension to the grammar in Figure 2.5 below.
2.2 Semantics

In this section we are going to explain the semantics associated with programs implementing components and systems consisting of a number of software components. We first discuss the semantics of component programs and subsequently the semantics of the parallel composition of component programs being the system program.

The semantics of a component program is defined as the set of sequences that correspond to the possible execution sequences of the basic statements, according to the program. We call the basic statements actions and the sequences traces. Our actions are atomic meaning that once an action starts executing, it completes without interruption. An atomic execution of two actions \( a \) and \( b \) is denoted by \( <a;b> \) with the interpretation that once action \( a \) starts executing, its execution is immediately followed by \( b \) without the possibility of interleaving with some other action. In other words the execution of \( a \) and \( b \) is contiguous, non-interruptable at any point.

**Figure 2.4.** Component program.

\[
C_i : \{ \text{ while (true) do } \ \\
\{ \ \\
1. \text{ receive}(fq_{i-1}, fPacket, 1); \\
2. \text{ receive}(bq_i, ePacket, 1); \\
3. \text{ process}\_fct(fPacket, ePacket); \\
4. \text{ send}(bq_{i-1}, fPacket, 1); \\
   \text{// fPacket became empty due to} \\
   \text{// memory recycling} \\
5. \text{ send}(fq_i, ePacket, 1); \\
   \text{// ePacket is full now} \\
\} \}
\]

**Figure 2.5.** Generative grammar specifying parallel composition.

\[
\begin{align*}
\text{SYSTEM} & \rightarrow '{'PC'}'. \\
\text{PC} & \rightarrow \text{PROGRAM} '||' \text{PC} \\
& | \text{PROGRAM}
\end{align*}
\]

2.2 Semantics
A trace is a finite or infinite sequence of symbols, where each symbol stands for an atomic action. Traces \( s \) and \( t \) can be combined by concatenation to \( s \cdot t \); if \( s \) is infinite this concatenation is just \( s \). Concatenation is generalized to sets of traces in the obvious way, by pointwise application. The length of a trace \( t \) is written as \( |t| \). The empty trace is denoted by \( \epsilon \). For trace \( t \), \( \text{Pref}(t) \) denotes the set of all prefixes of \( t \). We also denote the fact that a trace \( s \) is a prefix of \( t \) with \( s \subseteq t \).

Consider a program \( C \), and its associated alphabet \( A(C) \). The program is generated using the rules of the grammar presented in the previous section. We define function \( \text{Alph} \) that for each element in alphabet \( A(C) \) returns the corresponding set of atomic actions that appear in the program trace. We denote the set of atomic actions appearing in a trace by \( A'(C) \). Function \( \text{Alph} \) returns values from the power set of \( A'(C) \). Therefore:

\[
\text{Alph} : A(C) \rightarrow 2^{A'(C)}
\]

\[
\text{Alph}('(\text{skip}') \) def = \{\text{skip}\}
\]

\[
\text{Alph}('(\text{VAR} := \text{EXPR}') \) def = \{\text{VAR} := \text{EXPR}\}
\]

\[
\text{Alph}('(\text{receive} (\text{Q}, \text{VAR}, \text{NUMBER})') \) def = \{\text{Q}?\text{VAR}, \text{NUMBER}\}
\]

\[
\text{Alph}('(\text{send} (\text{Q}, \text{VAR}, \text{NUMBER})') \) def = \{\text{Q}!\text{VAR}, \text{NUMBER}\}
\]

\[
\text{Alph}('(\text{process}\_fct (\text{VAR LIST})') \) def = \{\text{c( VAR LIST) }\}
\]

\[
\text{Alph}('(G') \) def = \{G\}
\]

We denote with \( \text{Traces}(X) \) the set of all finite and infinite sequences over a set \( X \).

To define the semantics of a program we introduce function \( \text{Tr} \) that takes as input a program \( C \) and returns the set of traces that are possible according to the program. Hence:

\[
\text{Tr}(C) \subseteq \text{Traces}(A'(C))
\]

\( \text{Tr} \) is defined using induction over the grammar. The simplest program consists only of a basic statement. The trace set corresponding to such a program contains one trace, which consists only of the action in \( A'(C) \) corresponding to the basic statement. Thus we have:

\[
\text{Tr}('(\text{skip}') \) def = \{\text{skip}\}
\]

\[
\text{Tr}('(\text{VAR} := \text{EXPR}') \) def = \{\text{VAR} := \text{EXPR}\}
\]

\[
\text{Tr}('(\text{receive} (\text{Q}, \text{VAR}, \text{NUMBER})') \) def = \{\text{Q}?\text{VAR}, \text{NUMBER}\}
\]

\[
\text{Tr}('(\text{send} (\text{Q}, \text{VAR}, \text{NUMBER})') \) def = \{\text{Q}!\text{VAR}, \text{NUMBER}\}
\]

\[
\text{Tr}('(\text{process}\_fct (\text{VAR LIST})') \) def = \{\text{c( VAR LIST) }\}
\]
Note that we use the ‘\(<\)' and ‘\(>\)' symbols to denote the beginning and respectively the end of a trace. Also note that send and receive actions via a queue \(Q\) are denoted as \(Q!\) and \(Q?\) respectively like in CSP [Hoare, 1985]. Whenever the argument of these two actions is clear from the context we will use a shorthand notation in which the argument is dropped. The same holds for the trace action associated with the \texttt{process.fct} basic statement.

The trace set of a program consisting of a sequential composition of two statements \(S_1\) and \(S_2\) contains the concatenation of the traces corresponding to \(S_1\) and the traces corresponding to \(S_2\):

\[\text{Tr}(S_1; S_2) \overset{\text{def}}{=} \text{Tr}(S_1) \cdot \text{Tr}(S_2)\]

Another possible composition of two statements \(S_1\) and \(S_2\) is by using selection and a boolean expression (guard). We denote the guard with \(G\). The guard is evaluated and depending on the result either \(S_1\) or \(S_2\) is executed. The guard evaluation corresponds to an action in each trace where it occurs.

\[\text{Tr}(\text{if } (G) \text{ then } \{S_1\} \text{ else } \{S_2\}) \overset{\text{def}}{=} \{G\} \cdot \text{Tr}(S_1) \cup \{\neg G\} \cdot \text{Tr}(S_2)\]

Another way to compose a statement \(S\) and a guard \(G\) is by using repetition. For each iteration of the repetition the guard is evaluated and if the boolean value is \textit{true} the repetition body is executed, otherwise the execution stops. Before defining formally repetition, we introduce additional notation about traces. For \(T\) a set of traces,

\[ T^0 = \{\varepsilon\} \]
\[ T^{n+1} = T \cdot T^n, \quad n \geq 0 \]
\[ T^{\omega} = \{s|(\forall n : 0 \leq n : (\exists u, v : u \in T^n : s = u \cdot v))\} \]

Using this notation we define:

\[\text{Tr(while } (G) \text{ do } S) \overset{\text{def}}{=} (\cup_{n \geq 0}(\{G\} \cdot \text{Tr}(S))^n) \cdot \{\neg G\} \cup (\{G\} \cdot \text{Tr}(S))^{\omega}\]

Note that function \(\text{Tr}\) returns the set of \textit{potential} traces that record the \textit{potential} executions according to the component program. Actual executions will depend on the initial state (the initial values of program variables) in most cases, and the state upon evaluation of the guard. The situation where the actual execution is not influenced by the initial state is when the respective program variables are not present in the guard. We support this statement with two examples below. In all cases, a trace starting in an initial state \(s\) followed by a guard which evaluates to \textit{false}, is a \textit{potential} trace but is \textit{not} a trace that will record an \textit{actual execution}.

Consider the following example:
Program C₁ has the following corresponding trace set:

\[
\text{Tr}(C₁) = \{ \begin{array}{l}
\forall i := 0 \land i \geq 2 \\
\forall i := 0 \land i < 2 \land x := x + i \land i := i + 1 \land i \geq 2 \\
\forall i := 0 \land i < 2 \land x := x + i \land i := i + 1 \land i < 2 \\
\forall x := x + i \land i := i + 1 \\
\forall \omega > 1 \}
\end{array} \}
\]

The trace set returned by function \( \text{Tr} \) includes all traces that record the potential executions according to the component program. The initial state is partially fixed - variable \( i \) is initialized, but \( x \) is not. Nevertheless we know in advance which is the trace that records the actual execution because this trace is determined by the trace state \( s = \exists i := 0 \) preceding the evaluation of the guard, and the evaluation of the \( i < 2 \) guard. Obviously the trace recording the actual execution contains only two iterations of the while do loop regardless of the initial value of \( x \), because \( x \) is not present in the guard.

Also consider the case of a program containing a while(true) do statement, only the infinite repetition records the actual execution, regardless of the initial state. Consider the example below:

\[
C₂ : \{ i := 0; x := 0; \textbf{while}(\text{true}) \textbf{ do} \{ x := x + i; i := i + 1 \} \}
\]

The traceset corresponding to this program is

\[
\text{Tr}(C₂) = \{ \exists i := 0 \land x := 0 \land \text{false} \}
\]

The trace recording the actual execution of program \( C₂ \) does not contain any false evaluation of the guard:

\[
\exists i := 0 \land x := 0 \land (\text{true} \land x := x + i \land i := i + 1) \}
\]

In this case the initial state is completely fixed, both variables \( i \) and \( x \) are initialized at the beginning of the program. Indeed, because none of the variables are present in the guard, the initial state does not influence the number of loop iterations. Also note that there exists only one trace recording the actual execution because the logical value true returned at the evaluation of the guard cannot be changed from outside the component.

For the purposes of this thesis we will always fix the initial state by assuming initialization of all program variables. One reason is to be able to identify
the trace recording the actual execution. Another reason for imposing a unique initial state is because we wish to have a unique set of values for the program variables associated with a prefix of the trace. This way we are able to associate a state reached during execution with a partial execution of a program.

Why is it then relevant that we know all potential traces? Because in some situations, the environment in which a program executes can intervene in its execution as well. For example, we could have program $C_1$ executing concurrently with another program $C_4$. Program $C_4$, decrements the value of variable $i$ during each iteration of the loop of program $C_1$, thus canceling the effect of the $i$ increment in $C_1$. In this case, program $C_1$ executes an infinite loop and the trace recording this execution is infinite as well. In our semantics these possibilities will occur again as traces in the parallel composition. Other examples in which the environment intervenes such that any trace in the trace set records the actual execution of program $C_1$ are easy to find and we will not present them here.

We use the arguments presented above in deriving the trace set of component program $C_i$ in the TSSA chain presented in the previous section (2.4):

$$Tr(C_i) = \{ \forall \ false \ \exists, \\
(\forall \ true \ \neg f_{i-1} \ \neg b_i \ \neg c_i \ \neg b_{i-1}! \ \neg f_i! \ \exists)^\omega \}
$$

The component trace that records the actual execution is $(\forall \ true \ \neg f_{i-1} \ \neg b_i \ \neg c_i \ \neg b_{i-1}! \ \neg f_i! \ \exists)^\omega$. We denote this trace with $\text{CompTr}_i$ (1 ≤ $i$ ≤ $N$). All $\text{CompTr}_i$ (1 ≤ $i$ ≤ $N$) traces are infinitely repetitive and the actions that represent one iteration are $true$, $f_{i-1}$, $b_i$, $c_i$, $b_{i-1}$, and $f_i$ in the order they have been presented. We denote the trace that records one iteration of $\text{CompTr}_i$ with $t_{C_i}$ meaning that

$$\text{CompTr}_i = (t_{C_i})^\omega, (1 \leq i \leq N)$$

A specific iteration $k$ of $\text{CompTr}_i$ is denoted with $t_{C_i}^k$.

Note that for a system composed of $N$ components we use indices to identify the basic statements representing the processing body of a component in the program and also to identify the corresponding trace actions:

$$\text{Alph}'(\text{process}_{fct \_1} (\text{VAR LIST }')) \overset{def}{=} \{c_1(\text{VAR LIST })\}$$

$$\cdots$$

$$\text{Alph}'(\text{process}_{fct \_N} (\text{VAR LIST }')) \overset{def}{=} \{c_N(\text{VAR LIST })\}$$

Also

$$Tr'(\text{process}_{fct \_1} (\text{VAR LIST }')) \overset{def}{=} \exists c_1(\text{VAR LIST }\exists$$

$$\cdots$$
\[ Tr' (process_{\text{fct}}N (VAR\ LIST)) \overset{\text{def}}{=} c_N(VAR\ LIST) \]

Before discussing the parallel composition of programs we define the projection of a trace \( t \) to a certain alphabet \( A \), denoted by \( t \uparrow A \), as the trace obtained from \( t \) by removing all symbols not in \( A \) while maintaining the order given in \( t \). For trace \( t \) and symbol \( a \) we define the counting operator \( \# \) as follows:

\[ \#(t, a) = |t \uparrow \{a\}| \]

Informally, \( \#(t, a) \) denotes the number of occurrences of \( a \) in \( t \). When the \( t \) is clear from the context, we use a short-hand notation \( \#a \).

We arrived at the point where we can define the trace set given by the parallel composition of two component programs \( C_0 \) and \( C_1 \). We start with a simple example, \( C_0\|C_1 \), with \( C_0 : \{ x := a + b ; y := b - c ; \} \) and \( C_1 : \{ z := c \cdot a ; p := d / b \} \). The trace sets associated with each program are

\[
\begin{align*}
Tr(C_0) &= \{ x := a + b \; y := b - c \} \\
Tr(C_1) &= \{ z := c \cdot a \; p := d / b \}
\end{align*}
\]

The trace set of the parallel composition of the two programs contains all possible interleavings of the trace from \( Tr(C_0) \) with that of \( Tr(C_1) \):

\[
\begin{align*}
Tr(C_0\|C_1) &= \{ x := a + b \; y := b - c \; z := c \cdot a \; p := d / b \} \\
\end{align*}
\]

In general, the trace set yielded by the parallel composition of two programs \( C_0 \) and \( C_1 \) contains all possible traces constructed by arbitrarily interleaving the traces in \( Tr(C_0) \) with those from \( Tr(C_1) \):

\[
\begin{align*}
Tr(C_0\|C_1) &= \{ s \in Traces(A'(C_0) \cup A'(C_1)) \} \\
&= s \uparrow A'(C_0) \cup A'(C_1) = s \wedge \ \\
&= s \uparrow A'(C_0) \in Tr(C_0) \wedge s \uparrow A'(C_1) \in Tr(C_1).
\end{align*}
\]

Notice that because of the unique initial state, the trace set of a program that consists of only sequential composition contains a single trace. However in the case of the parallel composition of multiple programs, because of the arbitrary interleaving of the program traces, the trace set of the parallel composition contains many traces in spite of fixing the initial state for each of the programs. Also as a last observation, even in parallel composition of the component programs \( C_i \) in the TSSA chain, traces \( CompTr_i \ (1 \leq i \leq N) \) still record the actual execution. That is because the evaluation of the guard in the repetition of all component programs is not influenced by the values of any variable.
Moreover, even if that were not the case, none of the component programs interferes in the execution of another by changing the values of a variable during the repetition loop.

2.3 States, invariants and channels

A trace set corresponding to a component program represents all possible complete executions. A machine being in the middle of such an execution has only executed a prefix of such a trace. We associate such a prefix with a state during execution. Hence, the set of states during execution is characterized by the prefixes of the trace set, the prefix closure. For a trace set $T$ we denote this set of states by $\text{St}(T)$.

Properties that are true for all states of a trace set are called invariants. For example, with $C : \{ \textbf{while } (\text{true }) \textbf{ do } \{ B_1; B_2 \} \}$ we have

$$I : 0 \leq \#(t,b_1) - \#(t,b_2) \leq 1 \text{ for all states } t \text{ in } \text{St}(\text{Tr}(C)).$$

Note that we used the notations $b_1$ and $b_2$ for the actions corresponding to basic statements $B_1$ and $B_2$. This invariant merely states that actions $b_1$ and $b_2$ alternate in every (partial) execution. In such an invariant we drop the state argument from the function and simply say that

$$0 \leq \#b_1 - \#b_2 \leq 1$$

is an invariant of $C$(or of $\text{Tr}(C)$).

Invariant $I$ above is an example of a synchronization condition. Actions $b_1$ and $b_2$ are in this example synchronized by virtue of the program syntax. We call such an invariant a topology invariant. However, instead of looking at invariants that the trace set already has, we can also impose invariants. These represent limitations on the execution of the atomic actions. These imposed invariants then lead to a restriction to the subset of traces and corresponding states for which the invariants hold. As an example, consider once more $C_0 || C_1$ with:

$$C_0 : \{ \textbf{while } (\text{true }) \textbf{ do } \{ B_1; B_2 \} \} \text{ and } C_1 : \{ \textbf{while } (\text{true }) \textbf{ do } \{ B_3; B_4 \} \}.$$

We now decide that action $b_1$ represents a send action and $b_4$ a receive action on an unbounded channel. This interpretation leads to imposing the invariant $\#b_1 - \#b_4 \geq 0$. This means that in the trace set of the parallel composition certain traces are ruled out as an accepted behavior compared to the situation when none of the statements were used for communication. (Notice that the invariant must hold for all states derived from this set). Alternatively, we may decide that $b_1$ and $b_4$ form a synchronous channel as in CSP. This is captured in
the invariant that \( b_1 \) and \( b_4 \) always follow each other immediately in the trace. This limits the trace set again; notice that this effectively orders \( b_2 \) and \( b_3 \) as well. As another example, assume that we assign \( C_0 \) a higher execution priority than \( C_1 \). This translates into the invariant that no \( C_1 \) actions can precede \( C_0 \) actions.

Following up on this discussion we would like to impose communication via bounded queues. This is expressed in the following invariant

\[
0 \leq L(Q) \leq \text{Cap}(Q)
\]  

(2.1)

with

\[
\text{Cap}(Q) > 0.
\]

(2.2)

The initial situation is \( L_0(Q) \). Note that \( L(Q) = L_0(Q) + \#Q! - \#Q? \) for a state \( s \). Hence we extend the notation for the queue length to \( L(s, Q) \) so as to include the state \( s \) in which we count the number of packets in the queue. In the rest of the document we use the short hand notation \( L(Q) \) when the state to which we refer is clear from the context.

Equation (2.1) can be rewritten as

\[
0 \leq L_0(Q) + \#Q! - \#Q? \leq \text{Cap}(Q).
\]  

(2.3)

The introduction of constraints on the interleaved behavior also leads to the notion of blocking. Consider a constrained trace set \( T \) and a particular state \( s \) of the system, i.e., an element of the prefix closure \( \text{St}(T) \). Suppose also that the trace set is a result of a parallel composition \( C_0 || C_1 \). Any action \( b_i \) in \( A'(C_0) \) such that \( s \curvearrowleft b_i \uparrow A'(C_0) \) is a state of \( \text{Tr}(C_0) \) and \( s \curvearrowright b_i \) is a state of \( T \) is called a ready action of \( C_0 \) in state \( s \). If \( s \curvearrowleft b_i \) is not a state of \( T \) it is apparently not possible to execute \( b_i \) in the constrained set. We say that \( C_0 \) is blocked at \( b_i \) in state \( s \) of \( T \), denoted as "\( C_0 \ b_i \) (in \( s \) of \( T \))". In most cases both \( s \) and \( T \) are clear from the context and then we leave them out. When \( T \) is given, then in any state \( s \) we can divide the set of components into blocked components (\( B(s) \)) and ready-to-run components (\( RR(s) \)).

We close this section with adding a few more concepts relevant to our model. We define \( \text{Comp} \) as a function taking as argument an action and returning the component with the alphabet to which action \( b \) belongs:

\[
\text{Comp}(b) = C \equiv b \in A'(C)
\]

Furthermore, consider a trace \( t \) written as \( t = t_0 \prec b_1 \prec b_2 \prec t_1 \). If \( \text{Comp}(b_1) \neq \text{Comp}(b_2) \), then we say that a context switch occurs between \( \text{Comp}(b_1) \) and \( \text{Comp}(b_2) \) in state \( t_0 \prec b_1 \).

Finally, we define the number of context switches (NCS) function taking as argument a finite trace from a trace set \( T \), and returning the number of context
switches occurring in the trace:

\[ NCS : T \rightarrow \mathbb{N}, \]

\[ NCS(e) = 0, \]
\[ NCS(b) = 0, \]
\[ NCS(b_1 \leftarrow b_2 \leftarrow t) = \begin{cases} 
NCS(b_2 \leftarrow t) & \text{if } \text{Comp}(b_1) = \text{Comp}(b_2) \\
NCS(b_2 \leftarrow t) + 1 & \text{otherwise}
\end{cases} \]

2.4 The Streaming Pipeline

We are now going to analyze the behavior of the streaming system example described in Figure 2.2. As a first step we identify those traces that specify the system behavior, and we present the method for identifying these traces in this section. We start by considering the trace set containing all arbitrary interleavings of components actions, yielded by the parallel composition of components. On these traces we progressively impose conditions in the form of predicates. Each time a condition is imposed, it reduces the trace set to a new one, containing only those traces that satisfy the condition. The predicates imposed denote properties or characteristics of the system execution such as the communication via bounded buffers and the fixed priorities assigned to the components. In the end, the trace(s) that satisfy these conditions specify the system behavior.

2.4.1 Channel constraints

We define \( A \) to be the union of the sets of trace atomic actions corresponding to all components programs (\( A = \bigcup_{i=1}^{N} A'(C_i) \)), and \( A^\omega \) the set of all infinite traces which are formed from actions in the set \( A \). The set of traces that results from the interleaving of the component traces is called \( T_{il} \):

\[ T_{il} \overset{def}{=} \text{Tr}( \big| i \big| C_i ) \]

The relations expressed in equations (2.1), (2.2), and (2.3) hold for all queues of this system as well. From the syntax of the components we obtain the following topology invariants.

\[ 0 \leq \#f_{q_{i-1}?} - \#b_{q_{i-1}!} \leq 1, \quad 1 \leq i \leq N \quad (2.4) \]
\[ 0 \leq \#b_{q_i}? - \#f_{q_i}! \leq 1, \quad 1 \leq i \leq N \quad (2.5) \]

For convenience, we give names to these differences and call them \( x_i \) and \( y_i \) respectively. The four different values of this pair correspond to different states
or positions within trace \( t^C_i \). We re-write the equations above as

\[
0 \leq x_i, y_i \leq 1, \ 1 \leq i \leq N
\]  

(2.6)

As the operations on the queues are executed in the component program shown in Figure 2.4, \( x_i \) and \( y_i \) can take new values:

\[
x_i = 1 \land y_i = 0, \ after \ action \ f_{q_{i-1}}?, \ 1 \leq i \leq N. \quad (2.7)
\]

\[
x_i = 1 \land y_i = 1, \ after \ action \ b_{q_i}?, \ 1 \leq i \leq N. \quad (2.8)
\]

\[
x_i = 0 \land y_i = 1, \ after \ action \ b_{q_{i-1}}!, \ 1 \leq i \leq N. \quad (2.9)
\]

\[
x_i = 0 \land y_i = 0, \ after \ action \ f_{q_i}!, \ 1 \leq i \leq N. \quad (2.10)
\]

In general the following property holds:

**Property 2.1.** \( \text{Cap}(b_{q_i}) = L(f_{q_i}) + L(b_{q_i}) + x_{i+1} + y_i, \forall 0 < i < N. \)**

**Proof.**

\[
\begin{align*}
L(f_{q_i}) + L(b_{q_i}) &= \{ \text{the initial state of the queues is:} \ L(f_{q_i}) = 0 \land L(b_{q_i}) = \text{Cap}(b_{q_i}), \\
&\quad \forall 0 < i < N \} \\
&= \{ x_{i+1} = \#f_{q_i}? - \#b_{q_i}!, \ y_i = \#b_{q_i}? - \#f_{q_i}! \} \\
&\quad \text{Cap}(b_{q_i}) - x_{i+1} - y_i.
\end{align*}
\]

\[\Box\]

In the following we restrict ourselves to the channel-consistent traces of this system, i.e. those traces in \( T_{il} \) that for all their prefixes satisfy (2.1) for all channels in the system. Predicate \( \text{Scc} \) specifies this constraint. The set obtained by imposing \( \text{Scc} \) on all traces from \( T_{il} \) is called \( T_{cc} \):

\[
\text{Scc}(t) \overset{\text{def}}{=} (\forall s \in \text{Pref}(t), f_{q_i}, b_{q_i}. (1 \leq i < N): \\
0 \leq \#(t, f_{q_i}!) - \#(t, f_{q_i}?) \leq \text{Cap}(f_{q_i}) \land \\
0 \leq \text{Cap}(b_{q_i}) + \#(t, b_{q_i}!) - \#(t, b_{q_i}?) \leq \text{Cap}(b_{q_i})).
\]

\[
T_{cc} \overset{\text{def}}{=} \{ t \in T_{il} | \text{Scc}(t) \}.
\]

Imposing \( \text{Scc} \) limits the order in which actions can interleave, and introduces blocking. For the new constrained set \( T_{cc} \) we recapitulate the definitions for \textit{ready-to-run} and \textit{blocked} components. Given \( s \) from \( St(T_{cc}) \) a component \( C_i \) is \textit{ready-to-run} in state \( s \) when for an action \( a \) in \( A(C_i) \) such that \( s \models a \uparrow A(C_i) \) is a state of \( Tr(C_i) \) we have that \( s \models a \in St(T_{cc}) \). Also, \( a \) is called a \textit{ready} action of \( C_i \) in state \( s \). If \( s \models a \) is not an element of \( St(T_{cc}) \) it is not possible to execute \( a \) in the constrained set in which case we say that \( C_i \) is \textit{blocked} from channel perspective at \( a \) in state \( s \) of \( T_{cc} \).
Next we show that blocking at a send action in any state of $T_{cc}$ is not possible.

**Property 2.2.** For component $C_i (1 \leq i \leq N)$, blocking at $bq_{i-1}!$ is not possible in any state of $T_{cc}$.

*Proof.* We prove this by contraposition by considering a state $s$ where $C_i$ would be blocked at $bq_{i-1}!$ and showing this state cannot exist. To be blocked at $bq_{i-1}!$, execution of $C_i$ has to proceed until just before $bq_{i-1}!$. This means that for this state $s$ we have

$$x_i = 1 \quad (2.11)$$

In addition, blocking is caused by the channel consistency, hence $L(s, bq_{i-1}) = \text{Cap}(bq_{i-1})$. This expression is equivalent with

$$\text{Cap}(bq_{i-1}) + #(s, bq_{i-1}!) - #(s, bq_{i-1}?) = \text{Cap}(bq_{i-1}) \quad (2.12)$$

meaning that

$$#(s, bq_{i-1}!) - #(s, bq_{i-1}?) = 0 \quad (2.13)$$

We compare the situation between $C_i$ and $C_{i-1}$:

$$0 < \begin{cases} (2.11), (2.6) \\ x_i + y_{i-1} \end{cases}$$

$$= \begin{cases} (2.4), (2.5) \\ #(s, fq_{i-1}?!) - #(s, bq_{i-1}!) + #(s, bq_{i-1}?) - #(s, fq_{i-1}!) \end{cases}$$

$$= \begin{cases} (2.13) \\ L(fq_{i-1}) \end{cases}$$

$L(fq_{i-1})$ is at least 0 according to (2.1), which is a contradiction. □

By symmetric reasoning we also have:

**Property 2.3.** For component $C_i$, blocking at $fq_i!$ is not possible in any state of $T_{cc}$. □

From these two properties we conclude that when blocking of components due to communication occurs, it is possible only at input actions.

At the end of this section we include two more properties.

**Property 2.4.** Let $C_i$ be such that $C_i$ is blocked at action $fq_{i-1}? in s$ of $T_{cc}$ then $C_{i-1}$ cannot be blocked at action $bq_{i-1}? in s$.

*Proof.* We assume that $C_i$ is blocked at action $fq_{i-1}? in s$ of $T_{cc}$ and $C_{i-1}$ is blocked at action $bq_{i-1}? in s$. If $C_i$ is blocked at action $fq_{i-1}? then $L(fq_{i-1}) = 0$. Also $C_{i-1}$ blocked at action $bq_{i-1}? implies $L(bq_{i-1}) = 0$.

We have that:
\[
\begin{align*}
\text{Cap}(b_{qi-1}) &= \{\text{Property 2.1}\} \\
L(f_{qi-1}) + L(b_{qi-1}) + x_i + y_{i-1} &= \{(2.10), (2.7)\} \\
0 + 0 + 0 + 0 &= 0
\end{align*}
\]
which is impossible given that according to (2.2) the capacities of all queues are strictly positive. \(\square\)

**Property 2.5.** Let \(C_i\) be such that \(C_i\) is blocked at action \(f_{qi-1}\)? [in s of \(T_{cc}\)] and if \(C_{i-1}\) is blocked [in s of \(T_{cc}\)], then \(C_{i-1}\) is blocked at action \(f_{qi-2}\)? in s as well.

**Proof.** Results directly from Property 2.2, Property 2.3 and Property 2.4. \(\square\)

### 2.4.2 Precedence order constraints

Next we introduce an additional restriction on the traces of \(T_{cc}\) in the form of a priority assignment. We define priority as a function \(P\) that returns for each component a unique natural (2.14) number with the interpretation that a higher number means a higher priority.

\[\forall i, j \in \mathbb{N}, i \neq j \iff P(C_i) \neq P(C_j). \] (2.14)

The execution mechanism will select the next ready action of the component with the highest priority in the ready-to-run set. The invariant that specifies this mechanism is presented below:

\[\text{Scp}(t) \overset{\text{def}}{=} (\forall s \in St(t), a \in A, u \in Traces(A) \land t = s \downarrow a \downarrow u : P(\text{Comp}(a)) = \max_{C \in RR(s)} P(C)).\]

Limiting \(T_{cc}\) according to \(\text{Scp}\) gives \(T_{pc}\), the priority consistent traces:

\[T_{pc} \overset{\text{def}}{=} \{t \in T_{cc} | \text{Scp}(t)\}.\]

**Property 2.6.** \(T_{pc}\) has precisely one element.

**Proof.** We wish to prove that \(|T_{pc}| = 1\). In the first part of the proof we show that \(T_{pc}\) consists of at least one element. We prove that by construction according to the definition of \(\text{Scp}\), following the algorithm below:

\[\rho_0 := \varepsilon;\]

\[\rho_{n+1} := \left\{ \begin{array}{ll}
\rho_n \downarrow a, & \text{if } \exists a : P(\text{Comp}(a)) = \max_{C \in RR(\rho_n)} P(C) \\
\rho_n, & \text{otherwise}
\end{array} \right.\]
We consider \( \rho = \lim_{n \to \infty} \rho_n \). \( Scp(\rho_i) \) holds for all \( i \), and each \( \rho_i \) is a prefix of \( \rho \), hence \( Scp(\rho) \) holds as well. Therefore \( |T_{pc}| \geq 1 \).

In the second part of the proof we show by contraposition that \( |T_{pc}| \leq 1 \).

We assume:

\[ \exists \; t_1, t_2 \in T_{pc}, \; t_1 \neq t_2 : t_1 = s \langle a_i \rangle u \land t_2 = s \langle a_l \rangle w. \]

The fact that the priorities assigned to all components are unique (2.14) and given that \( P(\text{Comp}(a_i)) = P(\text{Comp}(a_l)) \), implies that \( \text{Comp}(a_i) = \text{Comp}(a_l) \). This implies that in state \( s \) there do exist two \textit{ready} actions of the same component, which is impossible according to the definition of \textit{ready} action. Therefore \( |T_{pc}| \leq 1 \).

The conclusions of the two parts of the proof (\( |T_{pc}| \geq 1 \) and \( |T_{pc}| \leq 1 \)) imply \( |T_{pc}| = 1 \). \( \square \)

We denote the unique trace in \( T_{pc} \) with \( \rho \) and we are interested in the occurrence of context switches in \( \rho \). Consider a context switch between components \( C_i \) and \( C_j \), i.e., \( \rho = s \langle a_i \rangle b_j \langle u \rangle (a_i \in A'(C_i) \land b_j \in A'(C_j)) \). If \( P(C_i) < P(C_j) \) we say that \( C_j \) preempts \( C_i \), we call this situation \textit{preemption} and the context switch occurs due to preemption. In the other case (\( P(C_j) < P(C_i) \)) we call this a \textit{context switch due to blocking}.

At the end of this section we make a last observation. All components in the systems studied here become \textit{ready-to-run} as soon as they have data in their input queue. For this reason we will call them from here on \textit{data-driven} components.

2.5 Approach summary

The model presented above is a fairly conventional interleaving model. Compared to regular trace semantics there are two major differences. First, we use the complete executions according to the syntax as the semantics, rather than the prefix closure. Secondly, we do not introduce any synchronization concepts in the syntax or the trace semantics. Instead we introduce the interpretation of atomic actions as well as execution policies by limiting attention to those traces that satisfy the interpretation. This makes the semantics simpler and the manipulation easier.

We consider a system consisting of a parallel composition of communicating components. The corresponding trace set is limited to those traces that satisfy the channel properties for all channels (section 2.4.1). We call this the \textit{channel-consistent} traces. On top of this set we impose priorities (section 2.4.2). This results in just a single trace for the system, which depends
2.6 Introducing timing constraints

In this section we present further characteristics of the systems we study, namely timing constraints. We introduce a system composed of $N$ components where $N - 1$ are data-driven and the last component $C_N$ has a periodic behavior. We call component $C_N$ a time-driven component. Such a chain is presented in Figure 2.7.

The aim is again to study the behavior of the system by focusing on its corresponding trace set. We maintain the approach presented in section 2.5. The trace set we focus on is limited to those traces that satisfy the choice of the atomic action order in the components, channel properties, timing properties.
2.6 Introducing timing constraints

(owing to the periodic behavior of $C_N$), and the priority assignment to components. We start by analyzing the trace set containing all traces corresponding to all possible arbitrary interleavings of the components actions ($T_i$), and progressively we impose invariants such that eventually we obtain the trace (and schedule) that specifies the system behavior.

Before we detail this approach in section 2.6.3, we present the syntax and trace set for the execution of component $C_N$ (section 2.6.1). We also introduce a few basic concepts related to timing such as computation time associated with actions, schedule functions and chain response time in section 2.6.2.

2.6.1 A time-driven component

The last component in the chain ($C_N$) has similar behavior as the rest with respect to the operations on the queues. However this component has periodic behavior meaning that any iteration $i$ of its loop is executed after $i - 1$ time intervals $T_N$. At the programming level we add a basic statement called $\text{delay}\_\text{until}(\text{VAR})$ where $\text{VAR}$ refers to an absolute time. We also extend the trace alphabet by defining $\text{Alph}(\text{delay}\_\text{until}(\text{VAR})) \overset{\text{def}}{=} \{d(\text{VAR})\}$. The informal interpretation of $\text{delay}\_\text{until}(\tau)$ is that, if invoked before $\tau$, it delays the execution of component $C_N$ until time is $\tau$. We also add the basic statement $\text{getTime}(\text{VAR})$ where $\text{VAR}$ refers again to an absolute time. The corresponding trace action in the trace alphabet is defined by $\text{Alph}(\text{getTime}(\text{VAR})) \overset{\text{def}}{=} \{\text{gt}(\text{VAR})\}$. Statement $\text{getTime}(\text{VAR})$ is reading the current time and storing it in $\text{VAR}$.

The syntax for the execution of component $C_N$ is presented in Figure 2.8. Variable $T_0^N$ contains the time at which $C_N$ starts its execution.

The trace set of component $C_N$ is

$$Tr(C_N) = \{ t_C^N \sim false, \quad t_C^N \sim (\overline{\nu} \text{true} \sim f_{q_{N-1}}? \sim b_{q_{N-1}}? \sim c_N \sim b_{q_{N-1}}! \sim f_{q_{N-1}}! \sim \quad \nu i := i + 1 \sim \overline{d(T_0^N + i * T_N)} \overline{\psi})^\omega \}. \quad (1)$$

where $t_C^N$ specifies the statements preceding the loop guard in the program:

$$t_C^N = \overline{\nu} \text{gt}(T_0^N) \sim i := 0 \overline{\psi}.$$  

We denote with $t_C$ the following trace:

$$t_C = \overline{\nu} \text{true} \sim f_{q_{N-1}}? \sim b_{q_{N-1}}? \sim c_N \sim b_{q_{N-1}}! \sim f_{q_{N-1}}! \sim \quad \nu i := i + 1 \sim \overline{d(T_0^N + i * T_N)} \overline{\psi}. \quad (2)$$

In practice in some cases, at the time that $C_N$ must start executing, an atomic action of another component executes on the CPU. Given the atomicity of the action, its execution cannot be preempted by $C_N$ even when $C_N$ has a
higher priority, and therefore $C_N$ is delayed until the end of the action execution. This delay is limited to a value $\mu$ which is a given system specific value. $\mu$ is at most the maximum over all computation times of atomic actions:

$$\mu \leq \max_{a \in \mathcal{A}'} \delta(a). \quad (2.15)$$

$\delta(a)$ represents the computation time of action $a$ and is defined in the next section. In practice the $c_i$ actions ($1 \leq i \leq N$) corresponding to the processing body of a component has in general a much larger computation time. For this reason we renounce to the atomicity of actions $c_i$ and we decompose it into a number of smaller actions of computation time. From this perspective our model resembles the model of tasks used for fixed priority scheduling with deferred preemption (FPDS). Indeed in the case of FPDS, each job of a task is assumed to consist of a number of non-preemptable subjobs. The tasks can only be preempted at the end of a subjob, and not during a subjob execution ([Bril, Lukkien & Verhaegh, 2007a], [Bril, Lukkien & Verhaegh, 2007b], [Burns, 1994]).

In view of the decomposition of action $c_i$ into smaller actions, we present below the new trace set associated both for a data-driven component as well as for a time-driven component. The corresponding trace actions in the trace alphabet of the new basic statements are defined below:

$$\begin{align*}
C_N : & \{ \text{getTime}(T_0) \\
i & := 0; \\
\text{while} \ (true) \ \text{do} \\
\{ \\
& \text{receive}(f_{qN-1}, fPacket, 1); \\
& \text{receive}(b_{qN}, ePacket, 1); \\
& \text{process}_fctN(fPacket, ePacket); \\
& \text{send}(b_{qN-1}, fPacket, 1); \\
& \text{send}(f_{qN}, ePacket, 1); \\
& i := i + 1; \\
& \text{delay}_\text{until}(T_0^N + i * T_N); \\
\} \\
\}
\end{align*}$$

Figure 2.8. Program of a time-driven component.

$$\begin{align*}
\delta(a) \text{ represents the computation time of action } a \text{ and is defined in the next section. In practice the } c_i \text{ actions } (1 \leq i \leq N) \text{ corresponding to the processing body of a component has in general a much larger computation time. For this reason we renounce to the atomicity of actions } c_i \text{ and we decompose it into a number of smaller actions of computation time. From this perspective our model resembles the model of tasks used for fixed priority scheduling with deferred preemption (FPDS). Indeed in the case of FPDS, each job of a task is assumed to consist of a number of non-preemptable subjobs. The tasks can only be preempted at the end of a subjob, and not during a subjob execution ([Bril, Lukkien & Verhaegh, 2007a], [Bril, Lukkien & Verhaegh, 2007b], [Burns, 1994]).}
\end{align*}$$
2.6 Introducing timing constraints

\[ \text{Alph}'(process \_ fct_N(VAR \ LIST)) \overset{\text{def}}{=} \{ c_N^1(VAR \ LIST), \ldots, c_N^{m_N}(VAR \ LIST) \} \]

\[ Tr'(process \_ fct_1(VAR \ LIST)) \overset{\text{def}}{=} \chi c_1^1(VAR \ LIST) \cap \cdots \cap c_1^{m_1}(VAR \ LIST) \chi, \]

\[ Tr'(process \_ fct_N(VAR \ LIST)) \overset{\text{def}}{=} \chi c_N^1(VAR \ LIST) \cap \cdots \cap c_N^{m_N}(VAR \ LIST) \chi. \]

From here on, given the fact that the arguments (VAR LIST) of the processing actions mentioned above do not change, we drop the arguments when we specify these actions in a trace. The new traceset associated to the program of a data-driven component \( C_i \) is:

\[ Tr(C_i) = \{ \chi false \chi, \]
\[ (\chi true \cap f_{q_{i-1}}? \cap b_{q_i}? \cap c_1^i \cap \cdots \cap c_{m_i}^{i-1} \cap f_{q_i}! \chi)^o \} \]

Also the associated trace set of a time-driven component \( C_N \) becomes

\[ Tr(C_N) = \{ t_N^0 \cap false, \]
\[ t_N^0 \cap (\chi true \cap f_{q_{N-1}}? \cap b_{q_N}? \cap c_N^1 \cap \cdots \cap c_N^{m_N} \cap b_{q_{N-1}}! \cap f_{q_N}! \cap i := i + 1 \cap d(T_N^0 + i \ast T_N) \chi)^o \} \]

where \( t_N^0 \) specifies the statements preceding the loop guard in the program:

\[ t_N^0 = \chi gt(T_N^0) \cap i := 0 \chi. \]

Trace \( t_N \) becomes:

\[ t_N = \chi true \cap f_{q_{N-1}}? \cap b_{q_N}? \cap c_N^1 \cap \cdots \cap c_N^{m_N} \cap b_{q_{N-1}}! \cap f_{q_N}! \cap i := i + 1 \cap d(T_N^0 + i \ast T_N) \chi. \]

2.6.2 Basic concepts related to timing

We introduce function \( \delta_p : A \times N \rightarrow N \) that given an input media stream \( p \) for each occurrence of an action from alphabet \( A \) in a trace \( t \) from \( T_t \), returns the computation time needed to execute it. The computation time is expressed in time units, e.g. CPU cycles. \( \delta_p(a, k) \) denotes the computation time of the \( k \)-th occurrence of action \( a \). Important to recognize is that the computation times of different occurrences of an action can be different. In the case of the media processing systems, the computation times of the actions are variable due to dependency on input stream contents. For the MPEG 2 streams the values of the function \( \delta \) can be determined by applying techniques shown in [Peng, 2000a].

We denote the \( k \)-th occurrence of an action \( a \in A'(C_i) \) in a trace with \( a^k \). From now on we take \( p \) fixed and, for ease of notation, do not write the dependence on \( p \) everywhere. Therefore for \( \delta_p(a, k) \) we use as a short hand notation
δ(a^k). Note that the kth occurrence of action a processes the kth packet in the input stream. The computation time of the delay_until action is 0. Lastly, we denote with S_δ(t) the function that returns the sum of computation times of actions in trace t and with S^M_δ(t) the sum of the worst case computation times of actions in t.

We introduce the schedule functions \( S_\sigma = \{ \sigma | \sigma : St(T_{il}) \rightarrow N \} \) of the concurrent execution of components \( C_i, i = 1..N \) on a processor. They capture decisions about when the actions of the components finish. For any state \( s \) from \( St(T_{il}) \), a schedule function returns the finishing time of state \( s \). The finishing time of a state from \( St(T_{il}) \) is expressed in units (such as the CPU cycles), hence the domain of values of the schedule functions is the set of natural numbers.

Once an atomic action starts executing, it completes without interruption. Hence, when in a given schedule \( \sigma \) a finite trace \( s \rightarrow a \) finishes at time \( \tau (\sigma(s \rightarrow a) = \tau) \), we know that execution of \( a \) started at time \( \tau - \delta(a) \) if \( a \) is not a delay action. Therefore, we can associate with each function \( \sigma \rightarrow a \) that indicates start times, which is perhaps a more natural interpretation of scheduling - it amounts to a time assignment of actions in a trace. We use our definition for the schedule functions because it is technically easier.

All time assignments in \( S_\sigma \) must satisfy a few properties. First, a time assignment must follow the monotonicity properties of real-time and in particular it must take the duration of action execution into account. This leads to the following soundness criterion:

\[
\sigma(s \rightarrow a) \geq \sigma(s) + \delta(a), \; \forall s \rightarrow a \in St(T_{il}), \; a \neq d(\tau) \quad (2.16)
\]

Secondly, when the platform switches execution between components there is a penalty to be paid in terms of a task switch time. This switching time depends on the current state of the system (captured in the trace until that point) and the new action. This consideration leads to a more precise formulation of (2.16), where \( Sw(s,a) \) denotes the time to switch context in state \( s \) to \( Comp(a) \).

\[
\sigma(s \rightarrow a) \geq \sigma(s) + \delta(a) + Sw(s,a), \; \forall s \rightarrow a \in St(T_{il}), \; a \neq d(\tau) \quad (2.17)
\]

A natural simplification is that \( Sw(s,a) \) depends solely on the last component active in \( s \). Then, when we write \( s = s \rightarrow b \), \( Sw(s,a) \) depends only on \( Comp(b) \) and \( Comp(a) \). When \( Comp(b) \) and \( Comp(a) \) are equal, \( Sw(s,a) = 0 \).

With these restrictions, the only freedom of a scheduling mechanism is to delay the execution of an action. The meaning of the criterion expressed in (2.18) is that when a delay action \( (d(\tau)) \) follows a state \( s \) \( (s \in St(T_{il})) \) at a time earlier than \( \tau \), the delay action advances the time until \( \tau \). When the delay action
2.6 Introducing timing constraints

follows $s$ at a time equal or later than $\tau$, the delay action has no effect.

$$\sigma(s \rightarrow d(\tau)) \geq \tau, \forall s \rightarrow d(\tau) \in St(T_{cc}) \quad (2.18)$$

For a given schedule $\sigma$, the time interval between the end of switching time and the start time of the subsequent action is called idle time.

In the remainder of this document we will restrict our attention to only those schedules that indicate each action to be executed as soon possible (eager schedules). We focus on these schedules because they reflect the realistic execution of a streaming chain on the physical platform. Hence, we define the eager schedules to return for every state $s$ of a trace $t$ the minimum of the values returned by all schedules (in $S_\sigma$) for state $s \in St(T_{il})$:

$$\sigma_{eager}(s) \overset{\text{def}}{=} \min_{\sigma \in S_\sigma} \sigma(s), \forall s \in St(T_{il})$$

The following properties show that the eager schedule is sound as well. By definition $\sigma_{eager}$ is unique and we have from (2.17)

**Property 2.7.** $\sigma_{eager}(s \rightarrow a) = \sigma_{eager}(s) + \delta(a) + Sw(s,a), \forall s \rightarrow a \in St(T_{il}), a \neq d(\tau)$, $\Box$

and from (2.18)

**Property 2.8.** $\sigma_{eager}(s \rightarrow d(\tau)) = \max(\tau, \sigma_{eager}(s)), \forall s \rightarrow d(\tau) \in St(T_{cc})$ $\Box$

In systems without timing the system trace is determined only by buffer data availability and by the priorities assigned to components. In systems with timing the system trace is influenced by delay statements as well. In both cases we consider the eager schedule of the resulting trace as the considered time assignment. Unless indicated differently, we use $\sigma$ to mean this eager schedule. Notice that the eager schedule may have idle time.

We define the response time of the chain (RTC) for a packet $k$ as the time counted from the moment that the packet starts being processed by the first action of component $C_1$ ($f_{q0}^k$) until the finish time of last action of $C_N$ that processes $k$ ($f_{qN}^k$). RTC depends on the trace that records the chain execution, the schedule associated with this trace, and the contents of packet $k$ which determines the computation times of each component action:

$$RTC : T_{il} \times S_\sigma \times N \rightarrow N,$$

$$RTC(t, \sigma, k) \overset{\text{def}}{=} \sigma(s \rightarrow f_{qN}^k) - \sigma(u \leftarrow f_{q0}^k) - \delta(f_{q0}^k),$$

with $u \leftarrow f_{q0}^k \subseteq s \rightarrow f_{qN}^k \subseteq t$. 

2.6.3 A Characterization of Chain Execution

Having introduced the basic concepts related to timing, we are ready to detail the approach outlined at the very beginning of section 2.6, regarding studying the behavior of our systems by focusing on its corresponding trace set.

The first restriction we impose on the initial trace set $T_{il}$, yields the channel-consistent traces of this system, i.e. those traces in $T_{il}$ that for all their prefixes satisfy (2.1) for all channels in the system. Predicate $Scc$ defined in section 2.4.1 specifies this constraint. The set obtained by imposing $Scc$ on all traces from $T_{il}$ is $T_{cc}$.

Next, we wish to ensure that we select only those traces where action $gt(T_{0}^N)$ records indeed the start time of the time-driven component $C_N$, which also should mark the beginning of its first loop iteration. To that end we must ensure that $C_N$ is blocked at action $gt(T_{0}^N)$ as long as no full packets have been produced in $fq_{N-1}$. This way $gt(T_{0}^N)$ will read the beginning time of the first execution of $C_N$ as it is intended. Otherwise, when $C_N$ has the highest priority in the system, $gt(T_{0}^N)$ can potentially read a time much earlier than the time when $C_N$ can begin its first loop iteration, due to the fact that $C_N$ might still be blocked from channel perspective. For this reason we introduce an additional predicate that selects only those traces in $T_{cc}$ which adhere to the above requirement:

$$
S_{cgt}(t) \overset{def}{=} (\forall s \in Pref(t), f_{q_i}, b_{i}(1 \leq i < N) : \#(s, f_{q_{N-1}}) - \#(s, gt(T_{0}^N)) \geq 0).
$$

The new predicate limits $T_{cc}$ to a new set $T_{cgt}$ which contains only those traces which satisfy $S_{cgt}$ called start time reading consistent traces.

As a next step, given the eager schedule function $\sigma$, we focus only on those traces in $T_{cgt}$ that satisfy the following predicate:

$$
S\sigma c(u) \overset{def}{=} (\forall s \in St(u), d(\tau) \in A'(C_N), v \in A^\omega : u = s \triangledown d(\tau) \triangledown v : \sigma(s) \geq \tau \lor Act(s) = \emptyset)
$$

where $Act(s)$ is the set of actions (other than the delay action) that are ready from channel perspective in state $s$.

Predicate $S\sigma c$ disqualifies a number of traces in $T_{cgt}$ by restricting the situations in which a state can be followed by a delay action. $S\sigma c$ imposes that a delay action $d(\tau)$ can follow a state $s$, only if $\sigma(s)$ is later in time than time $\tau$ specified in $d(\tau)$, or if there are no actions (other than delay actions) that are ready. $S\sigma c$ implies that a delay action can only execute at an earlier time than $\tau$ when there are no other actions that are ready. From here follows that in this case the time interval $\tau - \sigma(s)$ is idle time.

In general, for a system containing multiple time-driven components, $S\sigma c$
is defined as

\[ S_{\sigma c}(u) \stackrel{def}{=} (\forall s \in St(u), d(\tau) \in Del(s), v \in A^0 : u = s \downarrow d(\tau) \downarrow v : \sigma(s) \geq \tau \vee Act(s) = \emptyset) \land \tau = \min_{\theta} d(\theta) \in Del(s)) \]

where \( Del(s) \) is the set of delay actions that satisfy the requirements for a delay action to execute in state \( s \) as shown in Figure 2.9.

\[ \text{Figure 2.9. The delay actions of earlier times are executed first: } \tau_1 < \tau_2 < \tau_3. \]

The new constraint yields another trace set \( T_{\sigma c} \) that contains the schedule-consistent traces that satisfy predicate \( S_{\sigma c} \):

\[ T_{\sigma c} \stackrel{def}{=} \{ u \in Tcgt | S_{\sigma c}(u) \} \]

Predicate \( S_{\sigma c} \) implies that if an occurrence \( i + 1 \) of an action from \( A'(CN) \) follows a state \( v \in St(T_{\sigma c}) \) in a trace \( u \in T_{\sigma c} \) then \( \sigma(v) \geq T_0^N + i \ast T_N \). This is expressed as follows:

**Property 2.9.** \( (\forall v \in St(T_{\sigma c}), a \in A'(CN), v \downarrow a^{i+1} \in St(T_{\sigma c}) : \sigma_{\text{eager}}(v) \geq T_0^N + i \ast T_N). \)

We are ready now to define the notion of ready and blocked from time perspective for all actions of \( A'(CN) \). Any action \( a \) in \( A'(CN) \) such that \( s \downarrow a \uparrow A'(CN) \) is a state of \( T_{egt} \) and \( s \downarrow a \) is a state of \( T_{\sigma c} \) is called a ready (from time perspective) action of \( CN \) in state \( s \). In such a situation we also say that \( CN \) is ready-to-run from time perspective in state \( s \). If \( s \downarrow a \) is not a state of \( T_{\sigma c} \) it is apparently not possible to execute \( a \) in the constrained set. We say that \( CN \) is **blocked from time perspective** at \( a \) in state \( s \) of \( T_{\sigma c} \), denoted as \( CN \text{ bt } a \text{ in } s \text{ of } T_{\sigma c} \). We also say that component \( CN \) is **idle** in state \( s \). If component \( CN \) is blocked from time perspective in state \( s \) it belongs in \( B(s) \), even if from channel perspective it could execute. If \( CN \) is ready-to-run from time perspective it belongs to \( RR(s) \). Note that if \( CN \) is ready-to-run from time perspective in state \( s \), it is implicitly ready-to-run in \( s \) also from channel perspective.

Finally we introduce a last restriction on the traces of \( T_{\sigma c} \) in the form of a priority assignment to the components. Predicate \( Scp \) expressing this restriction was defined in section 2.4.2. Limiting \( T_{\sigma c} \) according to \( Scp \) gives \( T_{pc} \), the
priority consistent traces:

\[ T_{pc} \overset{\text{def}}{=} \{ t \in T_{sc} | Scp(t) \} \].

The approach described above is illustrated in Figure 2.10.

![Figure 2.10. Selecting the system execution trace for systems with timing constraints.](image)

**Property 2.10.** \( T_{pc} \) has precisely one element.

**Proof.** Identical with the proof presented for Property 2.6. \( \square \)

We denote the unique trace in \( T_{pc} \) with \( \rho \).

### 2.7 Summary

In this chapter we have introduced a number of basic concepts used further on in this thesis. We have also formally introduced the systems we study by specifying the syntax for the program text of the system components. This has been achieved by indicating the grammar that generates the component program texts and the grammar for the parallel composition of components.

The semantics associated to a program text is specified using traces. We wish to study the system behavior by focusing on the corresponding trace set. However before we can characterize the trace(s) that specify the system behavior, we need to identify them. In this chapter we have detailed our approach to identify these traces. We started by considering the trace set containing all arbitrary interleavings of components actions, yielded by the parallel composition of components. On these traces we progressively imposed conditions in the form of predicates. Each time a condition is imposed, it reduces the trace set to a new one, containing only those traces that satisfy the condition. The
predicates denote properties or characteristics of the system execution. In that sense the predicates we impose are mechanisms for the selection of only those traces that specify an execution with the characteristics and properties of our system execution. In the end, the traces that satisfy all conditions specify the system behavior.

Examples of system characteristics specified by predicates include the communication via bounded buffers and the fixed priorities assigned to the components for systems without timing constraints. In the case of systems with timing constraints an additional constraint is imposed to capture the periodic execution of some components.

The conclusion of the chapter is that both in pipelined systems with and without timing constraints, there exists a unique trace that specifies the system execution. A second relevant issue is that there exists a unique eager schedule associated with this trace and this schedule satisfies also the soundness criteria as they have been defined for all time assignments.
In this chapter we analyze the behavior of a pipelined system without timing constraints as described in Figure 2.2. All components in the system are data-driven and the program according to which they execute is presented in Figure 2.4. We analyze the system behavior by studying the trace \( \rho \) that records its actual execution (section 3.1). The process of identifying this unique trace has been presented in sections 2.4 and 2.5. The characterization of this trace has a number of consequences useful in practice because it allows us to derive criteria for system design that show how to reduce:

- the memory used by the system by minimizing the buffer capacities (section 3.2.1)
- the CPU time used by the system by minimizing the number of context switches occurring during the execution. Minimizing \( NCS(\rho) \) implies reducing the overhead associated with context switches (section 3.2.3)
- the length of the initial phase of the trace which helps reducing the chain response time for the packets arriving during this phase (section 3.2.2)
- the chain response time for all packets (section 3.2.4).
3.1 Characterization of the unique trace \( \rho \)

We characterize the single trace \( \rho \) as a function of the priority assignment. In the first sub-section we present two lemmas and a theorem that allow the calculation of the trace at design time. In the second sub-section we show how the theorem helps answer practical questions about the minimum sufficient amount of memory, calculating and optimizing the length of the initial phase sub-trace and minimizing the number of context switches.

Consider a component \( C_i \) that is a left local minimum in terms of priority:

\[
\forall j : j < i : P(C_j) > P(C_i).
\]

Whenever it is executing, the components to the left of it are blocked. However, the blocking can only occur at a single program statement (or trace action), according to Lemma 3.1. In words, Lemma 3.1 states that whenever actions of this local minimum component are executed, the components to the left of it are blocked at reading an empty packet from their right neighbors. The lemma specifies \( C_j b bq_j? \) [in \( s \) of \( T_{cc} \)] in order to clarify the blocking context given by \( T_{cc} \). Indeed the blocking is due to the fact that predicate \( Scc(s \rightarrow bq_j?) \) does not hold.

**Lemma 3.1.** Let \( C_i \) be such that \( \forall j : j < i : P(C_j) > P(C_i) \) and consider a state \( s \) of \( St(\rho) \) such that the next action after \( s \) in \( \rho \) is one of \( A(C_i) \). Then \( C_j b bq_j? \) [in \( s \) of \( T_{cc} \)] for all \( j < i \).

**Proof.** The fact that \( C_i \) is a left local minimum implies that when it executes all other components to the left of it must be blocked. The question is at which action these components are blocked. This lemma states that all components \( C_j \), to the left of \( C_i \) are blocked at action \( bq_j? \). We prove this fact by contraposition. Let \( C_j \) be the leftmost component that is blocked elsewhere. Property 2.2 and Property 2.3 already showed that blocking at output is not possible. The only possible actions where blocking due to channel communication is possible is either at action \( fqj-1? \) or \( bq_j? \). For \( C_1 \), blocking at \( fq_0? \) is not possible because of the cooperative environment hence, \( bq_1? \) is the only possible place to block. Therefore we must assume, \( j > 1 \) and \( C_j b fq_{j-1}? \). The fact that component \( C_j \) is blocked at action \( fq_{j-1}? \), means that \( C_j \) just finished the execution of action \( fq_{j-1}! \) (step \( \circ \) in the program loop shown in Figure 2.4). At this step, the full packet received from \( fq_{j-1} \) has been recycled in \( bq_{j-1} \) and the empty packet received from \( bq_j \) has been sent in \( fq_j \). This implies:

\[
x_j = 0
\]  

(3.1)

Since \( C_j \) is the leftmost component that is blocked elsewhere, it means that
3.1 Characterization of the unique trace $\rho$

$C_{j-1}$ is blocked at $bq_{j-1}$?. In this case, from (2.7) follows that

$$y_{j-1} = 0$$

(3.2)

The fact that $C_j$ is blocked at action $fq_{j-1}$? implies that the forward queue $fq_{j-1}$ is empty meaning that the queue length $L(fq_{j-1}) = 0$. Similarly $C_{j-1} b bq_{j-1}$? implies that the backward queue $bq_{j-1}$ is empty meaning that the queue length $L(bq_{j-1}) = 0$. From Property 2.1 we have that

$$Cap(bq_{j-1}) = L(fq_{j-1}) + L(bq_{j-1}) + x_j + y_{j-1}, \ \forall 1 < j \leq N.$$  

However this would imply that $Cap(bq_{j-1}) = 0$, which is a contradiction given that $Cap(bq_{j-1})$ is strictly positive. \hfill $\Box$

We have a symmetric lemma for $C_i$ as a component with a local right-minimum priority in a chain $(\forall j: j > i : P(C_j) > P(C_i))$.

**Lemma 3.2.** Let $C_i$ be such that $(\forall j: j > i : P(C_j) > P(C_i))$ and consider a state $s$ of $St(T_{pc})$ such that the next action after $s$ in $T_{pc}$ is one of $A'(C_i)$. Then $C_j b fq_{j-1}$? [in $s$ of $T_{cc}$] for all $j > i$.

**Proof.** Proof analogous with the one of Lemma 3.1. \hfill $\Box$

Lemma 3.1 and Lemma 3.2 form the key for understanding the behavior of the whole pipeline. Consider component $C_m$ with minimal priority in the entire chain $(P(C_m) = \min_{i=1..N} P(C_i))$. The behavior of $C_m$ is given by

$$\exists (true \sim fq_{m-1}? \sim bq_m? \sim c_m \sim bq_{m-1}! \sim fq_{m}!)^{m-1}$$

Lemma 3.1 and Lemma 3.2 give us that before the first execution of $fq_{m-1}$?, all components $C_j$ to the left of $C_m$ have become blocked at $bq_j$? and all components to the right of $C_m$ are blocked at $fq_{j-1}$?. The unique trace recording the interleaved execution of components up to the first execution of an action of $C_m$ is composed of

$$6 \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} Cap(bq_i) + 2(m - 1)$$

actions (see Property 3.2), executed in an order determined by the priority of the components left of $C_m$. We denote this trace by $t_{init}$ and we call it the *initial phase* in the execution of the system.

After $t_{init}$, a few actions of $C_m$ occur until $bq_{m-1}!$. Producing an empty packet in $bq_{m-1}$! de-blocks component $C_{m-1}$. $C_{m-1}$ executes one iteration of its loop during which it also produces an empty packet in $bq_{m-2}$, which in turn de-blocks $C_{m-2}$. In fact, in this manner all components $C_j$ to the left of $C_m$ are de-blocked in cascade. These components will execute one iteration of their loop and eventually return to being blocked again at $bq_j$?. The execution order
of the component actions is entirely determined by the component programs and the priority assignment. The trace recording this execution is of length \(6(m - 1)\) and we denote it with \(t_L\). We denote the set of components to the left of \(C_m\) with \(LSC_m\). At this point, \(C_m\) is the only \textit{ready-to-run} component in the chain, therefore it resumes its execution with action \(fq_m!\), according to its program. This results in de-blocking in cascade of the components \(C_j\) to the right of \(C_m\). These components will execute a full iteration until each component is blocked at the same action again (\(fq_{j-1}\)). This trace has length \(6(N - m)\) and we call it \(t_R\). We also denote the set of components to the right of \(C_m\) with \(RSC_m\). At this point \(C_m\) is again the only component \textit{ready-to-run} and the situation described above repeats. The order of actions in the traces \(t_L\) and \(t_R\) is always the same because the components programs and the priority assignment does not allow any deviation. In short this means that the system follows a repetitive behavior. We denote the trace that records this repetitive execution with \(t_{stable}\) and we call it the \textit{stable phase} in the execution of the system. We arrive at the following theorem.

**Theorem 3.1.** The pipeline system assumes a repetitive behavior after a finite initial phase. The complete behavior is characterized by

\[
\rho = t_{init} \searrow t_{stable}.
\]

\[
t_{stable} = (\exists \text{true} \searrow fq_{m-1}? \searrow bq_m? \searrow c_m \searrow bq_{m-1}! \kreuz \searrow t_L \searrow \exists fq_m! \kreuz \searrow t_R)^a
\]

where \(t_L\) and \(t_R\) are traces that record the interleaved execution of components in \(LSC_m\) and respectively in \(RSC_m\).

**Proof.** Follows directly from the above discussion. \(\square\)

The following corollary follows directly from Theorem 3.1:

**Corollary 3.1.** The stable phase starts when \(C_m\) executes for the first time. \(\square\)

At the end of this section we introduce the notion of system deadlock. We include here the definition of deadlock as presented in [Tanenbaum, 2001]. A set \(D\) of tasks with at least one not terminated is called deadlocked if all tasks in \(D\) are blocked, and for each non-terminated task in \(D\), any task that might unblock it is also in \(D\). System deadlock occurs when all tasks are in the deadlock set.

Property 3.1 shows a sufficient condition for the system to not deadlock.

**Property 3.1.** A sufficient condition so that deadlock does not occur is \(Cap(fq_i) \geq 1\) (and implicitly \(Cap(bq_i) \geq 1\), \(1 \leq i < N\).

**Proof.** We assume that the system is deadlocked and \(Cap(fq_i) \geq 1\), \(1 \leq i < N\). Implicitly \(Cap(bq_i) \geq 1\), \(1 \leq i < N\).
If the system is deadlocked, then at least one component \( C_j \) is blocked on communication with neighbor component(s). Property 2.2 and Property 2.3 show that blocking at send actions is not possible, therefore \( C_j \) can only be blocked at one of its receive actions. We consider for the beginning that \( C_j \) is blocked at \( f_{q_j-1} \), meaning that \( L(f_{q_j-1}) = 0 \). In this case \( C_j \) is blocked on communication with \( C_{j-1} \). \( C_{j-1} \) must be blocked as well (according to the definition of system deadlock).

If \( C_{j-1} \) were blocked at action \( b_{q_{j-1}} \), then according to (2.7), \( y_{j-1} = 0 \). This means that no empty packets are inside \( C_{j-1} \). This implies that all packets used for communication between \( C_{j-1} \) and \( C_j \) must be either in \( f_{q_{j-1}} \), or in \( b_{q_{j-1}} \), or inside \( C_j \). We have seen above that in this state \( L(f_{q_j-1}) = 0 \). Also no packets are inside \( C_j \), because \( C_j \) is blocked at action \( f_{q_j-1} \), meaning that \( x_i = 0 \) according to (2.10). This means that all packets available for communication between \( C_{j-1} \) and \( C_j \) must be in \( b_{q_{j-1}} \). Since \( \text{Cap}(b_{q_{j-1}}) \geq 1 \) follows directly that \( L(b_{q_{j-1}}) \geq 1 \), which implies that \( C_{j-1} \) cannot be blocked at action \( b_{q_{j-1}} \). Hence \( C_{j-1} \) must be blocked at action \( f_{q_{j-2}} \).

This reasoning continues until component \( C_1 \) which, because of the system deadlock, we also must assume to be blocked at receiving a full packet from \( f_{q_0} \). However this is impossible because the environment is cooperative and therefore blocking on the environment cannot happen.

A similar sort of reasoning can be applied if we consider that \( C_j \) is blocked at action \( b_{q_{j}} \).

### 3.2 Support for design practice

#### 3.2.1 Channel usage, minimizing channel capacities

The following corollary presents the status of all queues (with respect to the number of packets they store), just after \( C_m \) executes \( f_{q_{m-1}} \). This information becomes useful later when we address issues related to the chain response time for each packet in the chain (Section 3.2.4).

**Corollary 3.2.** Given state \( s \) in \( St(p) \)

\[
s = t_{\text{init}} \leftarrow (\langle \text{true} \rangle \leftarrow f_{q_{m-1}} \leftarrow b_{q_m} \leftarrow c_m \leftarrow b_{q_{m-1}} \leftarrow t_L \leftarrow f_{q_m} \leftarrow t_R) \right) \leftarrow (\langle \text{true} \rangle \leftarrow f_{q_{m-1}} \leftarrow t^j, j \geq 0),
\]

the following statements hold in \( s \):

a. \( \forall i : 1 \leq i < m : L(f_{q_i}) = \text{Cap}(f_{q_i}) - 1 \land L(b_{q_i}) = 0 \)

b. \( \forall i : m \leq i < N : L(f_{q_i}) = 0 \land L(b_{q_i}) = \text{Cap}(b_{q_i}) \).

**Proof.**

a. When \( C_m \) executes \( f_{q_{m-1}} \), components \( C_i \) (\( 1 \leq i < m \)) are blocked at \( b_{q_i} \) (Lemma 3.1). That implies that \( L(b_{q_i}) = 0 \). Equation (2.7) implies that at this
point in the execution of the components programs, there are $\text{Cap}(f_q) - 1$ full packets available for consumption. That is because given the order of actions in the individual traces of components, each $C_i$ executes first $f_{qi-1}$? before $b_{qi}$?, meaning that when $C_i$ becomes blocked at $b_{qi}$? it will have already executed $f_{qi-1}$?, which makes that each $f_q$ contains $\text{Cap}(f_q) - 1$ full packets and one packet is inside the component $C_i$.

b. Lemma 3.2 implies that when $C_m$ executes $f_{qm-1}$?, components $C_i$ ($m < i \leq N$) are blocked at action $f_{qi-1}$?. This implies $L(f_{qi-1}) = 0$. $L(f_{qi-1}) = 0$ and equation (2.10) imply that at this point during the execution of these components $L(b_{qi-1}) = \text{Cap}(b_{qi-1})$. Equation (2.10) shows that no empty packet is inside $C_i$ when it becomes blocked at $f_{qi-1}$? because again, given the order of actions in the individual traces of components, each $C_i$ executes first $f_{qi-1}$? before $b_{qi}$?.

**Corollary 3.3.** The minimum sufficient for all queues capacities to avoid deadlock is 1.

**Proof.** Follows directly from Property 3.1 and the fact that all queues capacities in the system are strictly positive. □

### 3.2.2 Optimizing the initial phase length

In this subsection we calculate the length of the initial phase and show how $t_{\text{init}}$ can be reduced and even eliminated (Corollary 3.4). Reducing the initial phase is important because designers assign budgets to the chain based on the resource requirements of the stable phase.

**Property 3.2.** The length of the initial phase in number of actions is

$$|t_{\text{init}}| = 6 \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} \text{Cap}(b_{qi}) + 2(m - 1)$$

**Proof.** The number of actions executed until the execution of the system reaches the stable phase is the number of actions executed by all $C_i$ ($1 \leq i < m$) until all their backward input queues ($b_{qi}$, $1 \leq i < m$) are drained. All these queues are drained after $6 \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} \text{Cap}(b_{qi})$ actions. After that, all $m - 1$ components $C_i$ ($1 \leq i < m$) will execute the guard action, action $f_{qi-1}$?, and then they become blocked at action $b_{qi}$? ($1 \leq i < m$). This happens in $2(m - 1)$ actions. That leaves us with the total number of actions for the initial phase:

$$|t_{\text{init}}| = 6 \sum_{j=1}^{m-1} \sum_{i=j}^{m-1} \text{Cap}(b_{qi}) + 2(m - 1).$$
Corollary 3.4. \(|t_{\text{init}}|\) can be reduced by decreasing \(m\) and by decreasing the capacity of the queues in the system.

Proof. Direct consequence of Property 3.2.

3.2.3 Optimizing the number of context switches

In this subsection we show that the priority assignment influences the number of context switches in \(\rho\) and we give insight in what is the appropriate priority assignment that minimizes \(NCS(t_{\text{init}})\) and \(NCS(t_{\text{stable}})\).

Theorem 3.1 implies a few facts about the context switches inherent to the execution of the system at stable phase:

Corollary 3.5. The context switches occurring due to blocking during the stable phase cannot be eliminated, and regardless of the priority assignment, the number of context switches due to blocking during one iteration of the stable phase is always \(N - 1\).

Proof. Indeed Lemma 3.1 and Lemma 3.2 show that each time \(C_m\) executes, all other \(N - 1\) components in the chain are blocked. Theorem 3.1 shows that regardless of the priority assignment to the components, each iteration of the stable phase starts with \(C_m\) which de-blocks in cascade all components to the left in the chain, after which it de-blocks in cascade all components to the right in the chain. During each iteration of the stable phase all components execute one iteration of their loop of their program after which \(N - 1\) components (\(G_m\) is excluded) become blocked until they are de-blocked in cascade again by \(C_m\) at the beginning of the next stable phase iteration. Therefore during one iteration of the stable phase \(N - 1\) context switches occur due to blocking. This discussion also demonstrates that the context switches due to blocking are inherent to the execution of this type of chain and therefore cannot be eliminated.

Corollary 3.6. One context switch due to preemption occurring during the execution of \(C_m\) cannot be eliminated during each iteration of the stable phase.

Proof. Follows directly from Theorem 3.1 that during each iteration of the stable phase \(C_m\) is preempted at least one time and at most two times. First time is when \(C_m\) executes \(b q_{m-1}\) and it de-blocks in cascade the components to the left in the chain. The second preemption occurs when \(C_m\) executes \(f q_m\), and de-blocks in cascade the components to the right in the chain. One of these two context switches can be avoided by choosing \(m = 1\) or \(m = N\). The other context switch due to these preemptions cannot be eliminated, it is inherent to
the execution of the system during the stable phase.

\[\square\]

**Theorem 3.2.** a. Minimum NCS\(t_{\text{init}}\) is achieved when \(P(C_1) = \min_{i=1..N} P(C_i)\).

b. The minimum NCS during one iteration of \(t_{\text{stable}}\) can be achieved either when:

\[\begin{align*}
(i) & \quad P(C_1) < P(C_N) < P(C_{N-1}) < \ldots < P(C_2), \\
(ii) & \quad P(C_N) < P(C_1) < P(C_2) < \ldots < P(C_{N-1})
\end{align*}\]

with \(\forall i : 1 \leq i < N - 1, \text{Cap}(fq_i) = 2\).

**Proof.**

a. Corollary 3.4 implies that for \(m = 1\), the initial phase is eliminated, therefore NCS\(t_{\text{init}}\) is 0.

b. Corollary 3.5 shows that regardless of the priority assignment, context switches occurring during the execution of components \(C_i(i \neq m)\) due to blocking cannot be avoided. In the case of \(C_m\), this component does not block but is preempted after both actions \(bq_{m-1}\) and \(fq_m\). Only one of these two context switches can be avoided by choosing \(m = 1\) or \(m = N\). The other preemption is inherent to the execution during the stable phase (Corollary 3.6). Hence the only context switches that can be eliminated are the ones due to preemption of components \(C_i(i \neq m)\). In the following we will show that when assigning priorities as shown at point b.(i) and (ii), all context switches due to the preemption of all components except \(C_m\) can be eliminated.

In case (i), the priority assignment implies that all context switches caused by preemptions are avoided in trace \(t_R\), each component executes one loop iteration of its program and blocks again at action \(fq_i\), \(1 < i < N\).

In case (ii) the priority assignment suggested implies that all context switches caused by preemptions are avoided in trace \(t_L\), each component executes one loop iteration of its program and blocks again at action \(bq_i\), \(1 < i < N\). Moreover, with this priority assignment, had the capacities of the queues been 1, additional context switches would have occurred due to the blocking on the input forward queues of components \(C_2, \ldots, C_{N-1}\) (Corollary 3.2) therefore the additional condition on the minimal length of queues must be imposed for case ii: \(\forall i : 1 \leq i < N - 1, \text{Cap}(fq_i) = 2\).

\[\square\]

**Corollary 3.7.** The minimum number of context switches occurring during one iteration of the stable phase is \(N\).

**Proof.** The minimum number of context switches corresponds to those context switches that are inherent to the system execution and therefore cannot be
eliminated. Corollary 3.6, Corollary 3.5 and Theorem 3.2 show that the only context switches that cannot be eliminated during the stable phase are those due to blocking and one context switch due to the preemption of $C_m$. Corollary 3.5 shows that the number of context switches due to blocking during one iteration of the stable phase is $N - 1$. Therefore the minimum number of context switches occurring during one iteration of the stable phase of the system is $N$.

\( \square \)

**Tradeoff**

1. The priority assignment suggested by Theorem 3.2 - ii implies that all context switches caused by preemptions were eliminated, at the cost of a longer initial phase.

2. The priority assignment suggested by Theorem 3.2 - ii. achieves a minimum $NCS(t_{stable})$ at the cost of more memory needed ($\forall i: 1 \leq i < N - 1$, $Cap(f_{q_i}) = 2$).

### 3.2.4 Optimizing chain response time

The following theorem shows the conditions under which the response time for a packet $k$ is optimized.

**Theorem 3.3.** $RTC(\rho, \sigma, k), k \in \mathbb{N}$ is minimal when $m = 1$ and $P(C_1) < P(C_2) < P(C_3) < \ldots < P(C_N)$. When $m > 1$, the response time of the chain during the stable phase ($RTC(\rho, \sigma, k), k \in \mathbb{N}$ where $k$ are packets arriving during the chain stable phase) is reduced by reducing the capacities of queues preceding $C_m$ or by decreasing the value of $m$.

**Proof.** Stream packets are inserted in the chain in order, are processed in order, and are outputted from the chain in order.

The actions that process packet $k$ are interleaved with actions that process:

- packets that already are in the chain when packet $k$ is introduced. We denote the sum of the computation times of the actions that process these packets to the end of the chain with $S_{\delta before k}$.

- packets that have been introduced in the chain after packet $k$. These packets are processed up to some component in the chain until packet $k$ is outputted from the chain. We denote the sum of the computation times of the actions that process these packets $S_{\delta after k}$.

We also denote with $S_{\delta k}$ the sum of the computation times of the actions that process packet $k$ from entering the chain until leaving it, and with $S_{sw}$ the total overhead introduced by the context switches that occur during the execution between action $f_{q_0}^{\delta k}$ and action $f_{q_N}^{\delta k}$. 

We can therefore express the chain response time for packet \( k \) as
\[
RTC(\rho, \sigma, k) = S_{\delta \; before \; k} + S_{\delta \; after \; k} + S_{sw}.
\]

The fact that the actions that process packet \( k \) are interleaved with other actions processing the packets mentioned above, makes the chain response time associated with a packet \( k \) longer. \( RTC(\rho, \sigma, k) \) is minimal when it contains only the actions that process packet \( k \), the number of actions processing packet \( k \) is minimal, and the overhead introduced by context switches is reduced to its minimum.

We will show now that when \( P(C_1) = \min_{i=1..N} P(C_i) \) and \( P(C_1) < P(C_2) < \ldots < P(C_N) \):

a. \( S_{\delta \; before \; k} = 0 \),
b. \( S_{\delta \; after \; k} = 0 \),
c. \( S_{\delta \; k} \) is minimal, and
d. \( S_{sw} \) is minimal.

a. Corollary 3.2 shows the status of all queues in the system after packet \( k \) is introduced in the chain by the execution of \(fq_0^k\):
\[
\forall i : m \leq i < N : L(fqi) = 0 \land L(bqi) = Cap(bqi).
\]
This means that for \( m = 1 \), when packet \( k \) is inserted in the chain, no other full packet is already stored in the system queues. This means that no actions will be executed to process packets already stored in the system queues when packet \( k \) is inserted in the system, therefore \( P(G_1) = \min_{i=1..N} P(C_i) \) implies \( S_{\delta \; before \; k} = 0 \).

b. Corollary 3.2 also implies for \( m = 1 \) that when a subsequent packet \( k + 1 \) is inserted in the chain, all forward queues are empty, meaning that packet \( k \) has already left the chain. This means that there are no packets arriving in the chain and being processed after \( k \) was inserted and before it leaves the chain. This implies \( S_{\delta \; after \; k} = 0 \).

c. \( S_{\delta \; k} \) is minimal when \( P(C_1) < P(C_2) < \ldots < P(C_N) \), because in this situation each component \( C_i \) takes the packet as input as soon as it has been produced by the predecessor \( C_{i-1} \), without waiting until the predecessor executes its guard again and becomes blocked at action \(fq_{i-2}^k\). In this priority setting the processing of the guards and blocking of the components occurs during each iteration of the stable phase after the packet has left the chain.

d. The number of context switches occuring during the trace that records the processing of packet \( k \) (during the execution between action \(fq_{N+1}^k\) and action \(fq_N^k\)) is \( N \), corresponding to \( N \) preemptions. This number cannot be reduced more within the afore mentioned trace. Indeed, by eliminating all \( N \)
3.3 Summary

In this chapter we have presented a model for the dynamic behavior of linear media processing chains executing in a cooperative environment. We express the behavior of the chain as a trace of the actions of the components that make up the chain. We have formally proven that the trace becomes repetitive (the stable phase) after a finite prefix (the initial phase) and we have shown that this trace can be calculated at design time. This approach allows the calculation and optimization of the capacities of the queues between components, of the preemptions in this trace (such as in the priority assignment \( P(G) < P(C_N) < P(C_{N-1}) < \ldots < P(C_2) \)) we would obtain \( N \) context switches again (\( N - 1 \) context switches due to blocking and 1 due to the preemption of \( C_1 \)). Additionally that would increase \( S_{\delta k} \) because in this situation the processing of the guards and blocking of the components occurs during each iteration of the stable phase before the packet has left the chain.

Worthy to note is that while indeed when considering the ascending priority assignment \( (P(C_1) < P(C_2) < \ldots < P(C_N)) \) during the entire iteration of the stable phase NCS is not minimal, within the trace that records the processing of packet \( k \) (during the execution between action \( f_{q_k}^k \) and action \( f_{q_N}^k \)), NCS is minimal.

The second statement of the theorem was that when \( m > 1 \), the response time of the chain during the stable phase \( (RTC(\rho, \sigma, k), k \in \mathbb{N} \text{ where } k \text{ are packets arriving during the chain stable phase}) \) is reduced by reducing the capacities of all queues preceding \( C_m \) or by decreasing the value of \( m \).

As we have explained above, packet \( k \) cannot be processed at the end of the chain before all other \( k - 1 \) packets have been processed. We know that processing of packets to the end of the chain happens only during the stable phase of the chain when all queues preceding \( C_m \) are filled to their capacity (Theorem 3.1, Lemma 3.1). It means that when packet \( k \) is arrives in the chain during the chain stable phase, already \( \sum_{i=1}^{m-1} Cap(f_{q_i}) - 1 \) packets are stored in the queues preceding \( C_m \) in the chain. Follows directly that by reducing the capacities of the queues preceding \( C_m \), the number of packets preceding packet \( k \) ia also reduced, hence \( S_{\delta \text{ before } k} \) is reduced, which implies that \( RTC(\rho, \sigma, k) \) is reduced as well. The same effect is achieved when maintaining the queue capacities but reducing \( m \). This way the number of queues preceding \( C_m \), and hence the number of packets preceding \( k \) is reduced. From here follows again that \( S_{\delta \text{ before } k} \) is reduced, which implies that \( RTC(\rho, \sigma, k) \) is reduced as well.

\( \square \)
initial phase, of the number of context switches, and of the response time of individual components and the entire chain.

The repetitive nature of the chain is an important property that also makes reasoning about composition of chains much easier. Designers need only to reason in terms of patterns of execution at the level of the chain instead of reasoning about the individual behaviors of components within the whole system. This approach also makes systems open in the sense that the effect of inserting (or withdrawing) components from a chain can be rigorously predicted and controlled.
In this chapter we analyze the behavior of linear chains that contain both data-driven and time-driven components. In contrast to the system studied in Chapter 3, the systems we analyze in this chapter have timing constraints, and quality of service constraints. The timing constraints are induced by the periodic execution of the time-driven components. The quality of service constraints come from the fact that the input and/or the output actions of the system must be executed at a certain rate. As an example from practice, in this context this means that the first and/or last time-driven component in the chain (a video digitizer or respectively audio/video renderer), must display the audio/video information at a predefined fixed rate adjusted to the human ear/eye. Any delay in displaying the information causes audio/video artefacts and the perceived quality of service (QoS) diminishes. The fact that the execution of the first and/or last time-driven component in a system have a direct impact on the system quality of service, shows the strict relation between the timing constraints of these systems and their quality of service constraints.

As in Chapter 3, the approach we take in characterizing the execution of these systems is to identify the dependencies in the executions of the components. We prove that a time-driven component in a chain where all the other components are data-driven, has the same influence on the overall execution.
of a chain as a data-driven component with minimum priority has on a chain composed of only data-driven components. This reduces the analysis of this time-driven system to be identical to that of the data-driven system in Chapter 3. In the case where a system contains two time-driven components, we show how the system can be designed such that all components develop a dependency on only one of the time-driven components. By eliminating the double dependency on the time-driven components we achieve a predictable system execution during which resource usage of memory and CPU are optimized.

One other important result of this chapter refers to CPU overload situations in which the time-driven component at the end of the chain misses its deadline for a number of periods. In these cases we show how to design the system such that there always exists an infinite suffix of the trace \( \rho \) during which the chain satisfies the quality of service requirements. The results of this analysis are relevant because they show how to trade memory for lower processing power when designing systems that experience high variations in computation times of tasks. The trade-off proposed is advantageous because the cost of an additional amount of memory is much lower than the cost of processing power when CPU is overprovisioned to accommodate computational peaks.

The chapter is organized as follows. In section 4.1 we present the case of a system composed of \( N - 1 \) data-driven components and ending with a time-driven component. Section 4.2 presents the behavioral analysis in the case where the last component executes according to the interlaced standard. The case of a system where the first component is time-driven executing according to the interlaced standard and the rest of the \( N - 1 \) components are data-driven is presented in section 4.3. Finally, in section 4.4 we show the behavioural analysis of a system encountered in practice. This is the case of a video-surveillance system where the first and last components are time-driven and the rest of \( N - 2 \) components are data-driven.

In all cases mentioned above, components can have variable computation times. Also in all cases the quality of service requirements of the systems are directly related to their timing requirements. We formally specify these requirements and we show the mechanisms to be employed at design time such that the quality of service requirements (and therefore the timing requirements) are satisfied. Aside of that we show again in each case how performance parameters such as response time of the chain, NCS, memory and CPU use can be optimized at design time of the chain.
4.1 A linear chain with a time-driven component at the end

In this section we study the case of a linear chain ending with a time driven component as described in section 2.6.1. The program of such a component is shown in Figure 2.8. In practice, time driven components are usually implemented as interrupt service routines issued by some hardware device. A few examples from the media processing domain include the video renderer and the audio renderer components executing on the TriMedia chip (Figure 4.1) [Philips, 1999]. In this particular case, the hardware devices are the Video Out Unit (VOU), and the Audio Out Unit (AOU). VOU and AOU transfer video and respectively audio data read from the memory SDRAM to the corresponding Video Out or Audio Out port. Both VOU and AOU issue interrupts periodically to the VLIW CPU at the end of each transmitted video image and respectively audio sample. The VLIW CPU updates the video (or audio data) pointers, with pointers to the next data. This is subsumed in the \texttt{send(fqN, ePacket, 1)} statement in the program of a time driven component as described in Figure 2.8. The clock with which the interrupts are issued is given either internally or externally by some other device. When the VLIW CPU updates the pointer to the next data to be transmitted, the new pointer value is used at the start of the next video or audio data.

The interrupts are maskable implying that nesting of interrupts is allowed. However, to ensure that the real time constraints for the execution of the video

![TriMedia Block Diagram](Figure 4.1)
and audio renderer components are met, only limited nesting of events is allowed. In that sense it is required that the execution of the video and audio renderer must be completed within a certain interval of time (flush time) that starts with the next period (Figure 4.2).

Figure 4.2. Execution of component $C_N$ cannot be scheduled before $T_0^N + k \times T_N$ and must finish before $T_0^N + k \times T_N + \text{flush}$. 

We express this property of a system (sub-)trace as follows:

$$SQoSc(t, \text{pref}, f_{q_{N-1}}, f_{q_{N}}, T_0^N, T_N) \overset{\text{def}}{=}$$

$$\forall s_1, s_2 \in St(t), \forall k \in \mathbb{N}, s_1 \sim f_{q_{N-1}}^k \subseteq s_2 \sim f_{q_{N}}^k \subseteq t :$$

$$\sigma(\text{pref} \sim s_2 \sim f_{q_{N}}^k) < T_0^N + k \times T_N + \text{flush} \land$$

$$\sigma(\text{pref} \sim s_1 \sim f_{q_{N-1}}^k) - \delta(f_{q_{N-1}}^k) \geq T_0^N + k \times T_N$$

where $\text{pref} \sim t \in St(\rho)$. $T_0^N$ is the time until the first packet is processed by $C_N$. We call this time start-up time. Trace $t$ specifies a (sub-)trace of the overall system execution that follows a prefix $\text{pref}$. The flush term refers to the flush time. $\sigma$ refers to the eager schedule for $\rho$. Our observation above implies that the execution start of component $C_N$ can only be delayed within the limits of the flush time. This delay occurs due to the fact that in some cases at the time that $C_N$ must start executing, an atomic action of another component executes on the CPU. Given the atomicity of the action, its execution cannot be preempted by $C_N$ despite the higher priority, and therefore $C_N$ is delayed until the end of the action execution. Here we make the observation that the flush time must also deal with the jitter induced by variable computation time of the atomic action that delays the execution start of $C_N$. This delay is limited to the value $\mu$ defined in (2.15). Therefore the flush time satisfies the following property:

$$\text{flush} > \mu + S_0(t_{CN}^e \sim t_{CN})$$

(4.1)

As defined in section 2.6.2, $S_0(t_{CN}^e \sim t_{CN})$ returns the sum of the computation times of actions in trace $t_{CN}^e \sim t_{CN}$, meaning the sum of the computation times of actions executed during one iteration of $C_N$. 
4.1 A linear chain with a time-driven component at the end

In fact \( C_N \) is a regular real-time periodic task where the activation time is \( T_0^N + k \cdot T_N \) and the deadline is \( T_0^N + k \cdot T_N + \text{flush} \). Given the fact that \( C_N \) is implemented in practice as an interrupt service routine we will reflect this implementation decision in our model by considering \( C_N \) to have the highest priority in the chain.

\[ \text{SQoSc}(t, \text{pref}, f_{q_{N-1}}^!, f_{q_N}^!, T_0^N, T_N) \]

specifies a timing requirement of the \( C_N \) components but it specifies also a Quality of Service (QoS) requirement. Indeed in practice if \( C_N \) is a video or an audio renderer and its execution (within the overall execution of the system) would not satisfy the property specified by \( \text{SQoSc} \), the human user would notice video or audio artefacts. This means that the Quality of Service provided by the system is lower, or that the QoS requirement of the system is not satisfied. For this reason the \( \text{SQoSc} \) predicate is pronounced as "Quality of Service requirement on a (sub-)trace \( t \) following prefix \( \text{pref} \)" and the value returned by predicate \( \text{SQoSc} \) applied on a (sub-)trace \( t \) shows whether the QoS requirement is satisfied on this trace or not.

We present below a general formal definition of the QoS requirement for a time-driven component:

\[ \text{SQoSc}(t, \text{pref}, a, b, T_s, T_p) \]

\[ \begin{align*}
\forall s_1, s_2 \in \text{St}(t), \forall k \in \mathbb{N}, s_1 \sim a^k \subseteq s_2 \sim b^k \subseteq t : \\
\sigma(\text{pref} \sim s_2 \sim b^k) < T_s + k \cdot T_p + \text{flush} \land \\
\sigma(\text{pref} \sim s_1 \sim a^k) - \delta(a^k) \geq T_s + k \cdot T_p
\end{align*} \]

To simplify making reference to a group of components in a chain, we introduce the following notations: \( \text{LSC}_i \) denotes the set of components preceding component \( C_i \) in a chain, and \( \text{RSC}_i \) denotes the set of components following component \( C_i \) in a chain. We denote with \( C_N \) the entire chain composed of \( C_1, C_2, \ldots, C_N \).

4.1.1 QoS requirements

The question experts in the media processing domain seek to answer is how can the QoS requirement be satisfied for the trace \( \rho \). Better even, is there a condition that can be imposed at design time of a media processing chain such that the quality of service requirement is satisfied. We start answering that question by analyzing the behaviour of the left subchain (\( \text{LSC}_N \)), composed of components \( C_1, C_2, \ldots, C_{N-1} \).

Subchain \( \text{LSC}_N \) is composed of only data driven components. As long as \( L(\delta_{q_{N-1}}) < \text{Cap}(\delta_{q_{N-1}}) \), \( \text{LSC}_N \) executes exactly the same as the chain composed of only data driven components presented in Chapter 3. Before \( \text{LSC}_N \) produces the first packet in \( f_{q_{N-1}} \), the subchain reaches its repetitive execution
(stable phase) as shown in Chapter 3. The stable phase trace is

\[(\forall \text{true} \leftarrow f_{q_{m-1}} \leftarrow b_{q_m} \leftarrow c_m \leftarrow b_{q_{m-1}}!x \leftarrow t_L \leftarrow f_{q_m}! \leftarrow t_R)^l,\]

where \(C_m\) is the component with minimum priority in \(LSC_N\). Trace \(t_R\) records the interleaved execution of components \(C_1, \ldots, C_{m-1}\), and \(t_L\) records the interleaved execution of \(C_{m+1}, \ldots, C_{N-1}\). \(l\) is a finite number of iterations of the stable phase of sub-chain \(LSC_N\) until \(f_{q_{N-1}}\) is filled to its capacity.

Once the stable phase is reached, each packet is produced in \(f_{q_{N-1}}\) at the end of one iteration of the stable phase. The iteration of the stable phase that processes packet \(k\) to the end of the subchain \(LSC_N\) (in \(f_{q_{N-1}}\)) is

\[(\forall \text{true} \leftarrow f_{q_{m-1}} \leftarrow b_{q_m} \leftarrow c_m \leftarrow b_{q_{m-1}}!x \leftarrow t_L \leftarrow f_{q_m}! \leftarrow t_R)^k\]

Note that this stable phase lasts only as long as \(f_{q_{N-1}}\) has not yet become filled to its capacity (\(b_{q_{N-1}}\) drained).

The sum of the computation times of actions during the \(k^{th}\) iteration of component \(C_i\), \(i < N\) is \(S_B(t_{C_i})\). We consider for the beginning fixed computation times of actions of \(C_i\). In this case, the sum \(S_B(t_{C_i})\) has the same value for any packet \(k\), therefore we drop the superscript \(k\) in \(t_{C_i}\). In Chapter 3 we show that during one iteration of the stable phase each component \(C_1, \ldots, C_{N-1}\) executes one iteration of its individual trace. The eager schedule does not allow for any unused time between the actions of data-driven components as those in \(LSC_N\). From here follows that the duration of one iteration of the stable phase of \(LSC_N\) is \(\sum_{i=1}^{N-1} S_B(t_{C_i})\).

In Chapter 3 we explain that each packet is produced in \(f_{q_{N-1}}\) at the end of each iteration of the stable phase of \(LSC_N\). From here follows that the processing time between the production of two consecutive packets \((k\) and \(k+1\)) in \(f_{q_{N-1}}\) is \(\sum_{i=1}^{N-1} S_B(t_{C_i})\) for two consecutive iterations uninterrupted by the consumption of a packet by \(C_N\) from \(f_{q_{N-1}}\) and \(\sum_{i=1}^{N} S_B(t_{C_i})\) otherwise. This means that the processing time between the production of two consecutive packets is always at most \(\sum_{i=1}^{N} S_B(t_{C_i})\). We denote this sum with \(S\).

\[S = \sum_{i=1}^{N} S_B(t_{C_i})\]  \(\text{(4.2)}\)

In the next subsection we show that after a finite prefix of \(\rho\), regardless of the priority assigned to \(C_N\),

\[S < T_N\]  \(\text{(4.3)}\)
4.1 A linear chain with a time-driven component at the end

ensures that $C_N$ always executes periodic, at time interval $T_N$. This would imply that if at design time of the chain it is ensured that the sum of the computation times of actions in one iteration of $C_1, \ldots, C_N$ is always smaller or equal than $T_N$, $SQoSc(\rho, \epsilon, f_{qN-1}, f_{qN}!, T_0^N, T_N)$ is also satisfied. We set out to show this while characterizing the trace $\rho$.

4.1.2 Stable phase characterization

Based on the constraints imposed by invariants $Scc, Spc$, the schedule constraints, and the imposed condition that $S$ is always smaller than $T_N(4.3)$, we characterize the trace $\rho$. Aside the fact that this characterization shows what are the sufficient conditions needed such that $\rho$ satisfies the QoS requirement, it also yields facts about the execution of the chain that allow the calculation of the trace $\rho$ at design time of the chain.

We have seen in the previous subsection that when the subchain $LSC_N$ produces the first packet in $f_{qN-1}$, it has already reached its stable phase as explained in Chapter 3. During this stable phase, when the component with minimum priority in $LSC_N (C_m)$ executes, all backward queues $b_i$, $1 \leq i < m$ are drained (Lemma 3.1, Chapter 3). We denote with $\rho_{m-1}$ the trace starting with the first execution of $C_1$, and ending with the execution of $f_{qN-1}$:

\[
\rho = \rho_{m-1} \circ s_{m-1},
\]

\[
\rho_{m-1} = u_{m-1} \circ f_{qN-1}
\]

Trace $\rho_{m-1}$ includes the initial phase of the $LSC_N$ sub-chain and the first iteration of its stable phase.

Next we analyze how many packets are put in $f_{qN-1}$ every period and how many are taken every $T_N$. We study the behaviour of the chain from the action following $\rho_{m-1}$ of each iteration of the stable phase of $LSC_N$. That means that the time between the production of two consecutive packets ($k$ and $k + 1$) in $f_{qN-1}$ during one period $T_N$ is $S$.

$S < T_N$ implies that during $s_{m-1}$, within one period $T_N$, on average $T_N/S$ packets are produced in $f_{qN-1}$. Even if we do not make any assumption about the priority of $C_N$, we can say that $C_N$ will execute (and thus consume a full packet from $f_{qN-1}$) at most once every $T_N$. This implies that every new period $T_N$ at least $T_N/S$ packets are produced in $f_{qN-1}$ and at most one packet is consumed. Hence every new period the length of $f_{qN-1}$ increases with at least $T_N/S - 1$ packets. This also implies that after maximum $\text{Cap}(f_{qN-1})/(T_N/S - 1)$ periods $T_N$, $f_{qN-1}$ will be filled to its capacity. We use the value $T_N/S$ in the real numbers domain as opposed to the same value rounded down to emphasize that even when $T_N/S < 2$, once every few periods, during one period $T_N$, $LSN_N$ produces 2 full packets in $f_{qN-1}$ while $C_N$ consumes only 1, implying steady growth in the queue length. It follows directly
from the formulas above that the smaller the difference between \( S \) and \( T_N \), the higher is the number of periods \( T_N \) until \( f_{qN-1} \) is filled to its capacity.

When \( f_{qN-1} \) is filled to its capacity, \( b_{qN-1} \) is drained, which implies that \( C_{N-1} \) becomes blocked on \( b_{qN-1} \). Let \( x_{N-1} \) denote the iteration number such that \( C_{N-1} \) is blocked for the first time. We denote with \( \rho_{N-1} \) the trace starting with the first action following prefix \( \rho_{m-1} \) and ending with the action \( f_{qN-2}^{2^{S_{N-1}}} \) after which \( C_{N-1} \) becomes blocked at \( b_{qN-1} \).

\[
\rho = \rho_{m-1} \triangleright \rho_{N-1} \triangleright s_{N-1}, \quad \rho_{N-1} = u_{N-1} \triangleright f_{qN-2}^{2^{S_{N-1}}}
\]

In the suffix following \( \rho_{N-1} \) (\( s_{N-1} \)), \( C_{N-1} \) is de-blocked (and therefore executes) only when \( C_N \) produces an empty packet in \( b_{qN-1} \), which happens at most once every \( T_N \). This implies in turn that \( C_{N-1} \) will consume now every \( T_N \) at most 1 full packet from \( f_{qN-2} \).

By repeating the reasoning above for the sub-chain composed of components \( C_1, \ldots, C_{N-2} \) we find that within maximum \( \text{Cap}(f_{qN-2}) \times (T_N/(S - S_{N-1}) - 1) \) periods \( T_N \), \( f_{qN-2} \) will be filled to its capacity and \( b_{qN-2} \) will be drained. We denote with \( \rho_{N-2} \) the trace starting with the first action following prefix \( \rho_{m-1} \triangleright \rho_{N-1} \) and ending with the action \( f_{qN-3}^{2^{S_{N-2}}} \) after which \( C_{N-2} \) becomes blocked at \( b_{qN-2} \). We denote with \( x_{N-2} \) the iteration number such that \( C_{N-2} \) is blocked for the first time.

\[
\rho = \rho_{m-1} \triangleright \rho_{N-1} \triangleright \rho_{N-2} \triangleright s_{N-2}, \quad \rho_{N-2} = u_{N-2} \triangleright f_{qN-3}^{2^{S_{N-2}}}
\]

The reasoning continues with all other components up to and including \( C_m \) and we define the subsequent \( \rho_i \) subtraces \((i = N - 3 \ldots m)\) in a similar way as shown above. We rewrite \( \rho \) as

\[
\rho = \rho_{m-1} \triangleright \rho_{N-1} \triangleright \rho_{N-2} \triangleright \cdots \triangleright \rho_m \triangleright s_m,
\]

\[
\rho_m = u_m \triangleright f_{qm-1}^{2^{s_{m}}}
\]

We denote with \( t_{init} \) the prefix \( \rho_{m-1} \triangleright \rho_{N-1} \triangleright \rho_{N-2} \triangleright \cdots \triangleright \rho_m \) and we call this prefix the initial phase of the entire chain \( CN \).

\[
\rho = t_{init} \triangleright s_m
\]

**Lemma 4.1.** When \( S < T_N \), then \( C_i \triangleright b_{q_i} \)? in \( t_{init} \) of \( \rho \), \( \forall i, 1 \leq i < N \).

**Proof.** We know that at the end of \( \rho_{m-1} \) backward queues \( b_{q_i} \), \( 1 \leq i < m \) are drained (due to \( LSC_N \) have been reaching its stable phase as explained in Chapter 3). In addition, the discussion above shows that at the end of each subtrace \( \rho_i \) \((i = N - 1 \ldots m)\) each backward queue \( b_{q_i} \) \((i = N - 1 \ldots m)\) is drained. This implies that at the end of \( t_{init} \) all backward queues in the chain are drained.

In the following theorem we show that after \( t_{init} \) the execution of the system
becomes repetitive and from this point of view stable. For this reason we rename \( s_m \) (the sub-trace following \( t_{\text{init}} \) in \( \rho \)) with \( t_{\text{stable}} \).

**Theorem 4.1.** When \( S < T_N \) and \( P(C_N) = \max_{i=1}^{N} P(C_i) \), the pipeline system in which the last component is time-driven, assumes a repetitive, periodic behavior after a finite initial phase. The complete behavior is characterized by

\[
\rho = t_{\text{init}} \bar{\sim} t_{\text{stable}}
\]

\[
t_{\text{stable}} = (t_{C_N} \bar{\sim} t_L \bar{\sim} d(T_0^N + i \cdot T_N))^\omega
\]

\[
t_{C_N} = \text{true} \bar{\sim} f_{qN-1} \bar{\sim} b_{qN} \bar{\sim} c_1^N \bar{\sim} c_2^N \bar{\sim} \ldots \bar{\sim} c_m^N \bar{\sim} b_{qN-1} \bar{\sim} f_{qN} \bar{\sim} c_i \bar{\sim} \ldots \bar{\sim} c_{qN-1} \bar{\sim} f_{qN} \bar{\sim} \bar{c}_{qN} \bar{\sim} \ldots \bar{\sim} c_2^N \bar{\sim} c_1^N \bar{\sim} b_{qN} \bar{\sim} f_{qN} \bar{\sim} c_i \bar{\sim} \ldots \bar{\sim} c_{qN-1} \bar{\sim} f_{qN} \bar{\sim} c_i \]

as illustrated in Figure 4.3.

- \( t_L \) is the trace recording the interleaved execution of components in sub-chain \( LSC_N \).
- During one iteration of \( t_{\text{stable}} \) all components execute one iteration of their loop. At the end of each iteration of \( t_{\text{stable}} \) components in \( LSC_N \) are blocked at \( b_{qN} \), \( 1 < i \leq N \).
- \( SQoSc(t_{\text{stable}}, t_{\text{init}}, f_{qN-1}, f_{qN}, T_0^N, T_N) \) holds.

**Proof.** According to Lemma 4.1, at the end of \( t_{\text{init}} \), all components in \( LSC_N \) are blocked at action \( b_{qN} \), \( 1 < i \leq N \), and therefore dependent on \( C_N \) to produce one empty packet. Follows directly that the only component that can execute after \( t_{\text{init}} \) is \( C_N \). So far trace \( \rho \) can be expressed as:

\[
\rho = t_{\text{init}} \bar{\sim} t_{C_N} \bar{\sim} u
\]

where \( t_{C_N} \) records the execution of \( C_N \) just before the \( delay \) action:

\[
t_{C_N} = \text{true} \bar{\sim} f_{qN-1} \bar{\sim} b_{qN} \bar{\sim} c_1^N \bar{\sim} c_2^N \bar{\sim} \ldots \bar{\sim} c_m^N \bar{\sim} b_{qN-1} \bar{\sim} f_{qN} \bar{\sim} c_i \bar{\sim} \ldots \bar{\sim} c_{qN-1} \bar{\sim} f_{qN} \bar{\sim} c_i \]

When \( C_N \) executes \( b_{qN-1} \) it de-blocks \( C_{N-1} \). However because \( C_N \) has the highest priority in the chain, \( C_{N-1} \) will not preempt \( C_N \). The execution of \( C_N \) continues according to its program until just before the \( delay \) action. Given the restriction imposed by predicate \( S\sigma_c \) presented in Chapter 2, a \( delay \) action can only be executed when no other regular actions are ready or the time has come, which here it is not the case (\( C_{N-1} \) is ready-to-run). This implies that after \( t_{C_N} \) the only actions that can follow are those of \( C_{N-1} \). When \( C_{N-1} \) executes \( b_{qN-2} \), \( C_{N-2} \) is de-blocked. In fact, ultimately all components in \( LSC_N \) are de-blocked in cascade (exact order of execution is determined by priority
(72)

assignment). All these components \( C_i, i < N \) will execute one iteration of their individual traces after which they become again blocked at the \( bq_i \) action, and must wait again for \( C_N \) to produce 1 empty packet in \( bq_{N-1} \). The interleaved execution of components in \( LSC_N \) is recorded by trace \( t_L \). At this point \( \rho \) can be expressed as:

\[
\rho = t_{\text{init}} \sim t_{C_N} \sim t_L \sim u
\]

Given the fact that \( S < T_N \) implies that \( \sigma(t_{\text{init}} \sim t_{C_N} \sim t_L) < T_0^N + l \times T_N \). At the end of \( t_L \) all components in \( LSC_N \) are blocked again, which implies that the delay action follows in the trace. Hence at this point \( \rho \) can be expressed as:

\[
\rho = t_{\text{init}} \sim t_{C_N} \sim t_L \sim d(T_0^N + l \times T_N) \sim u
\]

\( \sigma(t_{\text{init}} \sim t_{C_N} \sim t_L) < T_0^N + l \times T_N \) also implies that the effect of the delay action is to advance time until moment \( T_0^N + l \times T_N \). This implies that \( \sigma(t_{\text{init}} \sim t_{C_N} \sim t_L \sim d(T_0^N + l \times T_N)) = T_0^N + l \times T_N \) (4.4)

At moment \( T_0^N + l \times T_N \), \( C_N \) becomes ready-to-run from time perspective and all the other components are blocked at action \( bq_i \), \( i < N \), and therefore dependent again on \( C_N \) to produce one empty packet. This is a similar situation as the one at state \( t_{\text{init}} \), meaning that the execution of the system repeats in the manner as explained above. Therefore \( \rho \) can be expressed as:

\[
\rho = t_{\text{init}} \sim (t_{C_N} \sim t_L \sim d(T_0^N + i \times T_N))^{o_2}
\]

Trace \( \tilde{\sigma}(t_{C_N} \sim t_L \sim d(T_0^N + i \times T_N))^{o_2} \) is denoted with \( t_{\text{stable}} \) so we have:

\[
\rho = t_{\text{init}} \sim t_{\text{stable}}
\]

The repetitive execution expressed above and (4.4) show that for each iteration of \( C_N \) during \( t_{\text{stable}} \):

\[
\sigma(s_{1-fq_{N-1}^k}) - \delta(fq_{N-1}^k) \geq T_0^N + k \times T_N,
\]

\( \forall s_{1-fq_{N-1}^k} \in \text{St}(\rho), \ t_{\text{init}} \subseteq s_{1}, \forall k \in \mathbb{N} \)

Also given (4.1) we have that

\[
\sigma(s_{2-fq_{N}^k}) < T_0^N + k \times T_N + \text{flush}, \forall s_{2-fq_{N}^k} \in \text{St}(\rho), \ t_{\text{init}} \subseteq s_{2}, \forall k \in \mathbb{N}
\]

This implies that \( SQ\sigma_{t_{\text{stable}}, t_{\text{init}},fq_{N-1}^k,fq_{N}^k,T_0^N,T_N} \) holds.

Given that the sum of each iteration of components \( C_i, i \leq N \) is lower than \( T_N \) (\( S < T_N \)) implies that \( C_N \) becomes ready-to-run every \( T_N \) both from time and channel perspective. This makes the execution recorded in \( t_{\text{stable}} \) not only repetitive but also periodic. Note that during \( t_{\text{stable}}, C_N \) executes periodic (strictly at time interval \( T_N \)), regardless of its priority.

Important to note is that in this case, the time-driven component \( C_N \), even when it has the highest priority in the entire chain, has the same effect on the stable phase trace of the data-driven components in the chain, as a data-driven
4.1 A linear chain with a time-driven component at the end

Figure 4.3. Execution of system ending with a time-driven component. $C_N$ has the highest priority.

component with minimum priority has on the rest of the components in the case of a chain composed of only data-driven components. This observation is coined "time beats priority".

Note that if $C_N$ would not have the highest priority in the chain, trace $\rho$ would have been:

$$\rho = t_{\text{init}} \prec (\exists \text{true} \prec f_{q_{N-1}} ? \prec b_{q_N} ? \prec c^1_N \prec c^2_N \prec \ldots \prec c^m_{q_N} \prec b_{q_{N-1}} ! \prec t_{L_1} \prec \exists f_{q_N} ! \prec i := i + 1 \prec t_{L_2} \prec d(T_0^N + i * T_N) )^\omega$$

where $t_{L_1} \prec t_{L_2} = t_L$, and $t_L$ is the trace recording the interleaved execution of components in sub-chain $LSC_N$. When $C_N$ is assigned the lowest priority in the chain then $t_{L_2}$ is the empty trace $\epsilon$. The order of execution is again entirely determined by the priority assignment to the components.

Figure 4.4. Execution of system ending with a time-driven component. $C_N$ has the lowest priority.

The following corollary shows that if $C_N$ has the highest priority, then whenever $C_N$ executes during the stable phase, all the other components in the chain ($C_i, 1 \leq i < N$) are blocked at action $b_{q_i}$.
Corollary 4.1. Consider $t$ a prefix of trace $\rho$ such that $t_{init} \in St(t)$ and the next action after $t$ is $b_{N-1}^k$, $k > 0$. If $S < T_N$ and $P(C_N) = \max_{i=1}^{N} P(C_i)$ then $C_i$ $b$ $b_{qi}$ in $t$, for all $i$, $1 \leq i < N$.

Proof. Follows directly from Theorem 4.1. □

The lemma below states that under the condition that $S < T_N$ and $C_N$ is assigned the highest priority in the chain, $SQoSc(t_{init}, \epsilon, f_{q_{N-1}^k}, f_{q_N^k}, T_0^N, T_N)$ holds.

Lemma 4.2. $S < T_N \land P(C_N) = \max_{i=1}^{N} P(C_i) \Rightarrow SQoSc(t_{init}, \epsilon, f_{q_{N-1}^k}, f_{q_N^k}, T_0^N, T_N)$.

Proof. Consider $s_1, s_2$ such that $s_1 \sim f_{q_{N-1}^k} \subseteq s_2 \sim f_{q_{N-1}^k} \subseteq t_{init}$, $k > 0$.

Proving $\sigma(s_1 \sim f_{q_{N-1}^k}) - \delta f_{q_{N-1}^k} \geq T_0^N + k \star T_N$ is trivial by virtue of the delay action that precedes $f_{q_{N-1}^k}$ in the program trace of $C_N$.

At moment $T_0^N + k \star T_N$ at least $k$ productions of full packets have been completed in $f_{q_{N-1}^k}$ ($S < T_N$), hence $L(f_{q_{N-1}^k}) > 0$ at this time. This implies that at moment $T_0^N + k \star T_N$ $C_N$ becomes ready-to-run (given the availability of input in $f_{q_{N-1}^k}$ and the time). Since that $C_N$ has also the maximum priority in the chain, follows that $C_N$ will also execute (the latest at $T_0^N + k \star T_N + \mu$ because it may preempt the execution of components in $LSC_N$). The maximum priority of $C_N$ also implies that the execution of $C_N$ is uninterrupted until action $f_{q_N^k}$. Given (4.1) follows that $\sigma(s_2 \sim f_{q_N^k}) < T_0^N + k \star T_N + flush$. □

The corollary below states that the QoS requirement is satisfied for the entire execution of the chain (entire trace $\rho$) if at design time it is ensured that $S < T_N$ and $C_N$ is assigned the highest priority in the chain.

Corollary 4.2. $S < T_N \land P(C_N) = \max_{i=1}^{N} P(C_i) \Rightarrow SQoSc(\rho, \epsilon, f_{q_{N-1}^k}, f_{q_N^k}, T_0^N, T_N)$.

Proof. Follows directly from Lemma 4.2 and Theorem 4.1. □

When considering variable computation times for the actions of components, by imposing $S^M < T_N$ where

$$S^M = \sum_{i=1}^{N} S^M(t_C).$$

the stable phase is reached again and all lemmas and corollaries presented above hold as well, given that $P(C_N) = \max_{i=1}^{N} P(C_i)$. This means that again the trace $\rho$ satisfies the QoS requirement. Indeed by ensuring at design time that $S^M < T_N$ where here $S^M$ takes into account the worst case computation time for all actions of components, the reasoning and the conclusions about the
4.1 A linear chain with a time-driven component at the end

execution of the chain presented above hold again. Such a restriction in the
design of a chain can be verified by measuring the load on the processor of one
iteration of each component.

4.1.3 Practical Applications

Similarly as in Chapter 3, trace $\rho$ can be calculated by choosing in each state
the ready action of the component with the highest priority. Knowing the trace
allows to calculate the number of context switches, and the number of actions
needed to process a packet from input to the output of a chain. The eager
schedule of the unique trace can also be calculated provided that the computa-
tion times of each action processing an input stream are known. This renders
the start and response times for individual tasks and response time of the chain.

Important to note is that because $S^M < T_N$, regardless of its priority, com-
ponent $C_N$ has the same effect on the stable phase trace of this chain, as a
data-driven component with minimum priority has in the case of a chain com-
posed of only data-driven components (described in Chapter 3). Owing to this
fact, we find that corollaries addressing the stable phase of a chain composed
of only data-driven components (with $C_N$ having the lowest priority) hold in
this case as well. From here we deduce that:

- the minimum necessary and sufficient capacity of each queue in the
  chain is 1 (Corollary 3.3).
- the response time of the chain is to be calculated as shown in Chapter 3,
  in the case of a chain composed of only data driven components with
  $m = N$. Also follows directly that response time of the chain cannot
  be improved by assigning the minimum priority to $C_1$ as suggested in
  Chapter 3 because that does not change the influence of the time-driven
  component $C_N$.
- according to Theorem 3.3 the response time of the chain is reduced by
  reducing the capacities of queues preceding $C_N$.
- the number of context switches during the stable phase is minimal by
  assigning priorities as $P(C_1) < P(C_2) < \ldots < P(C_{N-1})$ and $Cap(f_{q_i}) = 2$
  $\forall i, 1 \leq i < N - 1$ (Theorem 3.2).

4.1.4 Modeling overload situations

In practice ensuring at design time that the QoS requirement is satisfied for
the entire execution of the chain (entire trace $\rho$) by guaranteeing that $S^M < T_N$, is
either too costly and pessimistic, or it implies reducing the functionality of the
chain. That is because if measurements reveal that $S^M < T_N$ does not hold,
either the processor speed has to be increased, or the number of components
in the chain needs to be reduced such that for the new $S^M$, the condition is satisfied.

In the first case, increasing the speed of the processor makes the system more costly. This solution is also pessimistic because the computation time of components actions that process media streams varies greatly depending on the contents of the input media stream. Experience shows that in general the computation times of component actions only occasionally reach peaks such as specified in $S^M$. The second alternative - reducing the number of components in the chain implies reducing the functionality of the chain, which is also not desirable.

Because of the above reasons we do not propose guaranteeing $S^M < T_N$ in order to satisfy QoS requirements. We choose for a "middle ground" solution in which the QoS requirements are relaxed, and guaranteed to be met during an infinite suffix of $\rho$, despite computation times peaks. This solution allows us not to impose at the design time of the chain a too strict restriction that accommodates the rare worst case computation times of components actions ($S^M < T_N$). We make a compromise and we propose chains are designed such that they satisfy the following constraint:

$$\sum_{i=k-M+1}^{k} S^i + E \leq M \ast T_N, \ M \in \mathbb{N}. \quad (4.5)$$

$S^k$ represents the time between the production of packets $k-1$ and $k$ in $f_{qN-1}$. We assume the following priority assignment:

$$P(C_{N-1}) = \max_{i=1}^{N} P(C_i) \text{ and } P(C_N) = \min_{i=1}^{N} P(C_i).$$

In this case the subchain $LSC_N$ is allowed to not produce a new packet in $f_{qN-1}$ every period, but within any $M \ast T_N$ periods of $C_N$ at least $M$ packets are produced in $f_{qN-1}$. Such a restriction can be verified at design time of the chain based on experience and measurements as we mentioned before.

We mentioned before that we relax the QoS requirement such that it specifies that after a finite prefix, there does exist an infinite suffix during which $SQoS$ is satisfied. This relaxation is specified by $SQoS_{\text{overload}}$ below:

$$SQoS_{\text{overload}}(\rho) \overset{\text{def}}{=} (\exists \text{pref}, t, \text{pref} \sim t = \rho : \quad SQoS(t, \text{pref}, f_{qN-1}?; f_{qN}?; T_0^N, T_N))$$
Lemma 4.3. Given that equation (4.5) is a property of the system execution, \( P(C_{N-1}) = \min_{i=1}^{N} P(C_i) \) and \( P(C_N) = \max_{i=1}^{N} P(C_i) \), then there exists a state \( t_{init} \) in which all forward queues are filled to their capacity.

Proof. The priority assignment assumed above implies that there exists a state \( t'_{init} \) in which all forward queues \( f_{qi}, 1 \leq i < N - 1 \) are filled to their capacity.

We wish to prove that there also exists \( t_{init} \in St(\rho), t'_{init} \in St(t_{init}) \) in which all forward queues are filled to their capacity.

After \( t'_{init} \) the system executes according to the pattern described in Theorem 3.1, producing 1 full packet in \( f_{q_{N-1}} \) at the end of each \( S^k \).

Let \( k = z \times M \). We have then that

\[
\sum_{i=1}^{k} S^i + z \times E = \sum_{i=1}^{z} \sum_{j=i \times M - M + 1}^{i \times M} (S^j + E) \leq \{(4.5)\} \sum_{i=1}^{z} M \times T_N = z \times M \times T_N = k \times T_N.
\]

In short,

\[
\sum_{i=1}^{k} S^i + z \times E \leq k \times T_N \tag{4.6}
\]

The number of consumed packets from \( f_{q_{N-1}} \) is \( k \) at \( T_0^{N} + k \times T_N \). The number of produced packets in \( f_{q_{N-1}} \) is \( k + z \times E / S^M \) at \( T_0^{N} + k \times T_N \), at least. \( S^M \) is a maximal value for \( S^k \), for instance \( M \times T_N \) according to (4.5).

When \( z \times E / M \times T_N > M \), or \( z \times E / M^2 \times T_N > 1 \), or \( z > M^2 \times T_N / E \) we have:

- at \( k = z \times M \), at least \( M \) packets are stored in \( f_{q_{N-1}} \), hence
- there exists a time \( \tau \leq k \times T_N \) such that at this time precisely \( M \) packets are stored in \( f_{q_{N-1}} \).

This means that for a finite buffer (of capacity \( M \)), this buffer will fill to its capacity after a finite time. \( \square \)

Lemma 4.4. Given that equation (4.5) is a property of the system execution, \( P(C_{N-1}) = \min_{i=1}^{N} P(C_i) \), \( P(C_N) = \max_{i=1}^{N} P(C_i) \), and \( \text{Cap}(f_{q_{N-1}}) = M \), then there
exist $s_i \in St(\rho)$, $i \in \mathbb{N}$ such that $\sigma(s_i) = \sigma(t_{init}) + i \times M \times T_N$ and $L(s_i, f_{q_{N-1}}) = M$.

**Proof.** We denote $\sigma(t_{init})$ with $T_{0}^{N'}$. The execution of the system is illustrated in Figure 4.5. Lemma 4.3 implies that at the end of $t_{init}$ all components in $LSC_N$ are blocked at action $b_{q_{i}}$, $1 \leq i < N$. We also have that

$$L(t_{init}, f_{q_{N-1}}) = Cap(f_{q_{N-1}}) = M.$$ 

and

$$L(t_{init}, b_{q_{N-1}}) = 0.$$ 

We want to calculate $L(f_{q_{N-1}})$ at the end of $M \times T_N$ periods after $T_{0}^{N'}$. For now, we express $L(f_{q_{N-1}})$ at the end of $M \times T_N$ periods as

$$L(f_{q_{N-1}}) = M + x_0.$$ 

During each of the next $M \times T_N$ periods, $C_N$ consumes 1 full packet from $f_{q_{N-1}}$ ($L(f_{q_{N-1}})$ is decremented) and produces 1 empty packet in $b_{q_{N-1}}$. This implies that after $M \times T_N$ periods $L(f_{q_{N-1}})$ is decreased with $M$ packets. Therefore we can express $x_0 = -M + x_1$ meaning that $L(f_{q_{N-1}})$ at the end of $M \times T_N$ periods becomes

$$L(f_{q_{N-1}}) = M - M + x_1.$$ 

Each time $C_N$ produces 1 empty packet in $b_{q_{N-1}}$, it allows the components in $LSC_N$ to be de-blocked in cascade. During each cascade de-blocking the components in $LSC_N$ execute one iteration of their loop after which they become blocked again at action $b_{q_{i}}$, $1 \leq i < N$. We denote with $t^k_{LSC_N}$ the trace recording the $k$ cascade de-blocking of components in $LSC_N$. Note that at the
end of each $t^k_{LSC_N}$, 1 full packet is produced in $fq_{N-1}$ ($CN_{-1}$ has the lowest priority in the chain therefore it executes last in $LSC_N$). Given that during each of the next $M*T_N$ periods, $CN$ produces 1 empty packet in $bq_{N-1}$, follows that $LSC_N$ is de-blocked $M$ times, and therefore $CN_{-1}$ produces $M$ packets in $fq_{N-1}$. Therefore we can express $x_1 = M$ meaning that $L(fq_{N-1})$ at the end of $M*T_N$ periods becomes

$$L(fq_{N-1}) = M - M + M = M.$$  

Note that each execution of $t^k_{LSC_N}$, $k = 1..M$ is characterized by

$$\sigma(u^{-1}t^k_{LSC_N}) < T_0^N + M*T_N, \ k = 1..M, \ u^{-1}t^k_{LSC_N} \subseteq \rho$$

and

$$\sigma(t^k_{LSC_N}) - S_8(t^k_{LSC_N}) \geq \sigma(v^{-1}t^k_{CN}), \ k = 1..M, \ v^{-1}t^k_{CN} \subseteq \rho.$$  

where $t^k_{CN}$ is the $k$ iteration of trace $t_{CN}$. This implies that each $t^k_{LSC_N}$, $k = 1..M$ can execute in a later period than the period in which the empty packet that de-blocks $LSC_N$ is produced by $CN$ but no earlier than that period. Also, it follows that $t^M_{LSC_N}$ ends before the end of the $M*T_N$ periods and that at that time ($T_0^N + M*T_N$), all components in $LSC_N$ will have executed one iteration of their loop and they are blocked again at action $bq_i$, $1 \leq i < N$, leaving again $CN$ the only ready-to-run component in the system. $CN$ executes its delay action which advances time until $T_0^N + M*T_N$. We denote the state that follows $t_{init}$ and ends with the delay action of $CN$ with $s_1$. Given the discussion above follows that state $s_1$ ends at moment $T_0^N + M*T_N$:  

$$\sigma(s_1) = \sigma(t_{init}) + M*T_N.$$  

To summarize, the above reasoning shows us there exists a state $s_1$ that ends at moment $\sigma(t_{init}) + M*T_N$ and that at the end of $s_1$  

$$L(s_1, fq_{N-1}) = Cap(fq_{N-1}) = M,$$

and

$$L(s_1, bq_{N-1}) = 0.$$  

and all components in $LSC_N$ are blocked at action $bq_i$, $1 \leq i < N$, with $CN$ the only ready-to-run component in the system, situation identical with the one at the end of $t_{init}$. This implies that this situation is repetitive hence the statement of the lemma holds at the end of each $T_0^N + i*M*T_N$, $i \in \mathbb{N}$.  

We denote the trace that follows $t_{init}$ in $\rho$ with $t^{overload}_{stable}$ with the mention that during $t^{overload}_{stable}$ the repetition consists only of the order of actions in $t^k_{LSC_N}$, $k = 1..M$ and the times at which $CN$ executes.
Corollary 4.3. Given that equation (4.5) characterizes the execution of the system, \( P(C_{N-1}) = \max_{i=1}^{N} P(C_i), P(C_N) = \max_{i=1}^{N} P(C_i) \), and \( \text{Cap}(f_{\overline{N}-1}) = M \), then \( SQoSc(t_{\text{load}}, t_{\text{init}}, f_{\overline{N}-1}, f_{\overline{N}}, T_{\text{init}}, T_{\text{load}}, T_{\text{N}}) \) holds and hence \( SQoSc_{\text{load}}(\rho) \) holds as well.

Proof. Results directly from Lemma 4.4.

4.2 The interlaced standard

In this section we present a different behaviour of the time driven component, as it appears in practice in the case of the visualization according to the interlaced standard. In this standard, video frames are displayed on the screen periodically, where every first period the even fields are displayed and every second period the odd fields are displayed. That means that every frame takes two periods of the time driven component to be displayed on the screen.

Component \( C_N \) receives one input full packet containing a decoded frame from \( f_{\overline{N}-1} \), separates the even from the odd fields and sends the odd fields and the even fields alternately, every second period. At the programming level, the basic statement \( \text{process}_N f_{cN}(\text{VAR,VAR}) \) denotes the processing during the first period of the time-driven component. \( \text{process}_N f_{cN}(\text{VAR,VAR}) \) denotes the processing during the second period of the time-driven component. We indicate in Figure 4.6 the program that specifies this behaviour.

The corresponding trace actions in the trace alphabet of the new basic statements are defined below:

\[
\begin{align*}
\text{Alph}'(\text{process}_N f_{cN}(\text{VAR LIST}')) & \equiv \{c_{N}'^{1}, \ldots, c_{N}'^{m_{N}'}\} \\
\text{Alph}'(\text{process}_N f_{cN}(\text{VAR LIST}')) & \equiv \{c_{N}'^{1}, \ldots, c_{N}'^{m_{N}'\prime}\}
\end{align*}
\]

The trace associated with each basic statement is

\[
\begin{align*}
\text{Tr}'(\text{process}_N f_{cN}(\text{VAR LIST}')) & \equiv \ulcorner c_{N}'^{1} \dashv \ldots \dashv c_{N}'^{m_{N}'\prime} \urcorner \\
\text{Tr}'(\text{process}_N f_{cN}(\text{VAR LIST}')) & \equiv \ulcorner c_{N}'^{1} \dashv \ldots \dashv c_{N}'^{m_{N}'\prime} \urcorner.
\end{align*}
\]
4.2 The interlaced standard

Figure 4.6. Program of a time-driven component $C_N$. Implementation according to the interlaced standard.

The associated trace set of component $C_N$ is

$$Tr(C_N) = \{ t^1_{C_N} \prec false, \\
t^2_{C_N} \prec (\not\prec true \prec \\
\quad fq_{N-1} \prec \neg bq_N \prec \neg c_1^N \prec \ldots \prec c_{m_1}^N \prec \neg fq_N \prec \\
copyfPacket = fPacket \prec \neg bq_{N-1} \prec \\
i := i + 1 \sim d(T_0^N + i \ast TN) \prec \\
bq_N \prec \neg c_1^{N''} \prec \ldots \prec c_{m_2}^{N''} \prec \neg fq_N \prec \\
i := i + 1 \sim d(T_0^N + i \ast TN) \not\sim \},$$

where $t^1_{C_N}$ specifies the statements preceding the loop guard in the program:

$$t^1_{C_N} = \not\prec gr(T_0^N) \prec i := 0 \not\prec .$$

As previously, in the trace recording the actual execution of $C_N$ the while guard value is true. For denotational purposes we consider as illustrated in Figure 4.6:

$$t^1_{C_N} = \not\prec true \prec fq_{N-1} \prec \neg bq_N \prec c_1^N \prec \ldots \prec c_{m_1}^N \prec \neg fq_N \prec \\
\quad copyfPacket = fPacket \prec \neg bq_{N-1} \prec i := i + 1 \not\prec ,$$

$$t^2_{C_N} = \not\prec bq_N \prec \neg c_1^{N''} \prec \ldots \prec c_{m_2}^{N''} \prec \neg fq_N \prec i := i + 1 \not\prec$$

In the following paragraphs we study the influence of this new behaviour
on the execution of the entire chain. We revisit the theory presented in the previous sections and we show in what way the previous findings change, and why. We consider for the beginning fixed computation times for all components actions.

We make the observation that the procedure of selecting the trace(s) which specify the overall execution of the chain does not depend on the individual traces of the components. Therefore this process remains the same as described in section 2.6.3 and the conclusion applies here as well, that there exists a unique trace that represents the system behavior.

4.2.1 QoS requirements

Next we examine the new QoS requirements brought up by this new behaviour of C_N and the mechanisms used to satisfy these requirements. In general the QoS requirement for any time-driven component that executes according to the interlaced standard is re-defined as

$$S\text{QoS}\text{i}(t, \text{pref}, a, b, c, d, T_s, T_p) \overset{\text{def}}{=} \begin{cases} (\forall s_1, s_2, s_3, s_4 \in St(t), \forall k \in \mathbb{N}, \\ s_1 - a^k \subseteq s_2 - b^k \subseteq s_3 - c^{k+1} \subseteq s_4 - d^{k+1} \subseteq t : \\ \sigma(\text{pref} - s_2 - b^k) < T_s + k * T_p + \text{flush} \land \\ \sigma(\text{pref} - s_1 - a^k) - \delta(a^k) \geq T_s + k * T_p \\ \sigma(\text{pref} - s_2 - d^k) < T_s + (k+1) * T_p + \text{flush} \land \\ \sigma(\text{pref} - s_1 - c^k) - \delta(c^k) \geq T_s + (k+1) * T_p) \end{cases}$$

where \( \text{pref} \in St(p) \). The parameters have the following meaning: \( a, b, c, d \) are the receive and respectively send actions executed during each of the two parts of the interlaced execution of a time-driven component (\( a \) and \( b \) correspond to the first part, \( c \) and \( d \) to the second); \( T_s \) denotes the execution beginning time of the time-driven component and \( T_p \) its period. The QoS requirement specified above imposes that each part of the interlaced execution of a time-driven component must be executed at a constant rate \( T_p \) (within the \text{flush} time).

In particular for \( C_N \) the QoS requirement is specified as follows:

$$S\text{QoS}\text{i}(t, \text{pref}, f q_{N-1}, f q_N, T_0^N, T_N) \overset{\text{def}}{=} \begin{cases} (\forall s_1, s_2, s_3, s_4 \in St(t), \forall k \in \mathbb{N}, \\ s_1 - f q_{N-1}^{t+k} \subseteq s_2 - f q_N^{t+k} \subseteq s_3 - b q_N^{t+k+1} \subseteq s_4 - f q_N^{t+k+1} \subseteq t : \\ \sigma(\text{pref} - s_2 - f q_N^{t+k}) < T_0^N + k * T_N + \text{flush} \land \\ \sigma(\text{pref} - s_1 - f q_{N-1}^{t+k}) - \delta(f q_{N-1}^{t+k}) \geq T_0^N + k * T_N \land \\ \sigma(\text{pref} - s_4 - b q_N^{t+k+1}) < T_0^N + (k+1) * T_N + \text{flush} \land \\ \sigma(\text{pref} - s_3 - b q_N^{t+k+1}) - \delta(b q_N^{t+k+1}) \geq T_0^N + (k+1) * T_N) \end{cases}$$
where $pref \tau \in St(\rho)$ and

$$flush > \mu + \max(S^M_0(t_{C_N}^1) + S^M_0(t_{C_N}^2), S^M_0(t_{C_N}^3))$$

(4.7)

When $SQoSc(\rho, \varepsilon, f_{q_{N-1}}?, f_{q_{N}!}, b_{q_{N}}? , f_{q_{N}!}, T_{N}^0, T_{N})$ holds, $C_N$ executes at a constant rate $T_{N}$, within the flush time interval (Figure 4.7).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.7}
\caption{Component $C_N$ executing according to the interlaced standard.}
\end{figure}

### 4.2.2 Characterization of system behaviour

We have shown in section 4.1.2 that a first mechanism designed to ensure the QoS requirement (imposing $S < T_{N}$ at chain design time) guarantees that whenever $C_N$ is ready to run from time perspective (which is strictly every $T_{N}$), it is ready also from channel perspective. This means that once $C_N$ starts executing, whenever $C_N$ must execute $f_{q_{N-1}}?$ (which is strictly every $T_{N}$), $f_{q_{N-1}}$ is never empty.

Given that according to the new program of $C_N$, a new full packet is needed to be consumed only once every $2T_{N}$, we can relax the $S < T_{N}$ condition and impose $S < 2T_{N}$ instead. Indeed the only difference between the previous behaviour and the new one in terms of the rate of performing operations on the $f_{q_{N-1}}$ and $b_{q_{N-1}}$ queues, is that instead of receiving/sending packets in these queues at rate $T_{N}$, it is doing it at rate $2T_{N}$. For this reason, all results regarding the status of these queues and all other queues derived from the fact that $S < T_{N}$ will hold now for $S < 2T_{N}$.

In this way we find again as in Lemma 4.1 that when $S < 2T_{N}$ there exists a finite prefix of $\rho$, $t_{init}$, at the end of which all components $C_i$, for all $i, 1 \leq i < N$, are blocked at action $b_{q}?$.

**Lemma 4.5.** $S < 2T_{N} \Rightarrow \exists t_{init} \in St(\rho): C_i b_{q}?[in t_{init} of \rho ], \forall i, 1 \leq i < N.$

$S < 2T_{N}$ has also an influence on the overall behaviour of the chain. As in Theorem 4.1, the Stable Phase Theorem below characterizes this behaviour in
terms of the trace of actions executed by the components in the chain. Trace $\rho$ is different here due to the different trace of component $C_N$. The proof of the theorem explains in more detail the nature and cause of differences.

![Diagram](image.png)

Figure 4.8. Execution of a system ending with a time-driven component. The time-driven component executes according to the interlaced standard.

**Theorem 4.2.** When $S < 2T_N$ and $P(C_N) = \max_{j=1}^{N} P(C_j)$, the pipeline system in which the last component is time-driven and implements the interlaced standard, assumes a repetitive, periodic behavior after a finite initial phase. The complete behavior is characterized by

$$
\rho = t_{\text{init}} \sim t_{\text{stable}}
$$

$$
t_{\text{stable}} = (\Rightarrow \{t_{C_N}^1, t_{L_1}, d(T_0^N + i \cdot T_N)\} \sim t_{C_N}^2, t_{L_2}, d(T_0^N + i \cdot T_N)) \Rightarrow
$$

as illustrated in Figure 4.8. Also:

- $t_{L_1}$ and $t_{L_2}$ are the traces recording the interleaved execution of components in sub-chain $LSC_N$.
- Sub-trace $t_{L_1}$ records the interleaved execution of components in $LSC_N$ between $t_{C_N}^1$ and $d(T_0^N + i \cdot T_N)$.
- Sub-trace $t_{L_2}$ records the interleaved execution of components in $LSC_N$ between $t_{C_N}^2$ and $d(T_0^N + i \cdot T_N)$ (of the next iteration $i$) when $T_N < S < 2T_N$. Trace $t_{L_2}$ is empty when $S < T_N$.
- During one iteration of $t_{\text{stable}}$ all components execute one iteration of their loop. At the end of each iteration of $t_{\text{stable}}$ components in $LSC_N$ are blocked at $bq_{i}$, $1 < i \leq N$. 
4.2 The interlaced standard

- SQoS\textsubscript{Sci}(t_{\text{stable}}, t_{\text{init}}, f_{Q_{N-1}}, f_{Q_{N}}, b_{Q_{N}}, f_{Q_{N}}, T_{0}, T_{N}) holds.

Proof. According to Lemma 4.5, at the end of \( t_{\text{init}} \), all components \( C_j \), for all \( j, 1 \leq j < N \), are blocked at action \( b_{Q_j} \). Due to this all components in the chain are dependent on \( C_N \) to produce one empty packet. Follows that the only component that is ready-to-run from channel perspective in the chain in this state is \( C_N \). However \( C_N \) becomes ready-to-run only at moment \( TN + l \ast TN \). Therefore state \( t_{\text{init}} \) is followed as of moment \( TN + l \ast TN \) by actions of \( C_N \) according to its program:

\[
t^{1}_{C_N} = \# true \triangleright f_{Q_{N-1}} \triangleright b_{Q_N} \triangleright c^1_{N} \triangleright \ldots \triangleright c^\mu_{N} \triangleright f_{Q_{N}} \triangleright b_{Q_{N-1}} \triangleright \\
i := i + 1 \\
\]

At this point the system trace can be expressed as:

\[
\rho = t_{\text{init}} \triangleright t^{1}_{C_N} \triangleright \nu.
\]

Also note that

\[
\sigma(t_{\text{init}} \triangleright s_1 \triangleright f_{Q_{N-1}} \triangleright) - \delta(f_{Q_{N-1}} \triangleright) \geq T_{0} + l \ast TN,
\]

and given that the execution of \( C_N \) is not preempted due to its maximum priority in the chain and due to (4.7), we also have:

\[
\sigma(t_{\text{init}} \triangleright s_2 \triangleright f_{Q_{N}} \triangleright) < T_{0} + l \ast TN + \text{flush}
\]

where \( s_1 \triangleright f_{Q_{N-1}} \triangleright \subseteq s_2 \triangleright f_{Q_{N}} \triangleright \subseteq t_{\text{stable}} \).

When \( C_N \) produces an empty packet in \( b_{Q_{N-1}} \) by executing \( b_{Q_{N-1}} \) it de-blocks component \( C_{N-1} \). The next action of \( C_N \) according to its program is the \textit{delay} action \( d(T_{0} + (l + 1) \ast TN) \)). However the \textit{delay} action cannot be executed at this point because now there exist other actions (of \( C_{N-1} \)) than \textit{delay} that are ready in this state. Therefore the next actions in the trace are those of \( C_{N-1} \).

When component \( C_{N-1} \) executes \( b_{Q_{N-2}} \) it de-blocks \( C_{N-2} \), which postpones the execution of the \textit{delay} action again. In fact the system experiences a de-blocking in cascade of components in \( LSC_{N} \), from \( C_{N-1} \) down to \( C_1 \). We distinguish the following case analysis:

A. When \( T_{N} < S < 2T_{N} \):

At time \( T_{0} + (l + 1) \ast TN + \mu \), given that \( C_{N} \) has the maximum priority in the chain, \( d(T_{0} + (l + 1) \ast TN) \) is executed. The additional \( \mu \) time comes from the fact that \( C_{N} \) preempts the execution of another component only after the atomic action that was executed at time \( T_{0} + (l + 1) \ast TN \) is completed. The trace recording the interleaved execution of components in \( LSC_{N} \) between the execution end of \( t^{1}_{C_N} \) and execution start of \( d(T_{0} + (l + 1) \ast TN) \) is \( t_{L1} \). At this point the system trace can be expressed as:
\[ \rho = t_{\text{init}} \overset{1}{\sim} \tau_{\mathcal{C}_N} \sim t_{L1} \sim d(T_0^N + (l + 1) \ast T_N) \sim u. \]

Because \( T_N < S < 2T_N \), follows that \( \sigma(t_{\text{init}} \overset{1}{\sim} \tau_{\mathcal{C}_N} \sim t_{L1}) / gEqT_0^N \geq + (l + 1) \ast T_N. \)

This means that action \( d(T_0^N + (l + 1) \ast T_N) \) does not have any effect in this case. Given that at moment \( T_0^N + (l + 1) \ast T_N \) \( C_N \) is ready-to-run, \( C_N \) continues its execution according to its program (Figure 4.6) with \( \tau_{\mathcal{C}_N}: \)

\[ t_{\mathcal{C}_N}^2 = \exists \ bq_N ? \sim c_N^1 ? \sim \cdots \sim c_N^n ? \sim bq_N ! \sim i := i + 1 \exists. \]

Note that in this case, the execution of \( C_N \) is not preempted due to its maximum priority in the chain and due to (4.7), we also have:

\[ \sigma(t_{\text{init}} \overset{1}{\sim} s_3 \sim bq_N^{i+1} \sim T_0^N + (l + 1) \ast T_N), \quad (4.10) \]

and given that the execution of \( C_N \) is not preempted due to its maximum priority in the chain and due to (4.7), we also have:

\[ \sigma(t_{\text{init}} \overset{1}{\sim} s_3 \sim bq_N^{i+1} \sim T_0^N + (l + 1) \ast T_N + flush) \quad (4.11) \]

where \( s_1 \sim bq_{N-1}^i \subseteq s_2 \sim bq_{N}^i \subseteq s_3 \sim bq_{N}^{i+1} \subseteq s_4 \sim bq_{N}^{i+1} \subseteq t_{\text{stable}}. \)

Trace \( \rho \) can be expressed as:

\[ \rho = t_{\text{init}} \overset{1}{\sim} \tau_{\mathcal{C}_N} \sim t_{L1} \sim d(T_0^N + (l + 1) \ast T_N) \sim t_{\mathcal{C}_N}^2 \sim u. \]

Note that in this case, the execution of \( C_N \) preempts the cascaded execution of components in \( LSC_N \). The delay action following \( \tau_{\mathcal{C}_N} \) according to the program of \( C_N \) cannot be executed yet because at the end of \( \tau_{\mathcal{C}_N} \) there exist actions of components in \( LSC_N \) other than delay that are ready in this state.

Therefore the cascaded execution of components in \( LSC_N \) is resumed and recorded by \( t_{L2} \). At the end of \( t_{L2} \), all components in \( LSC_N \) have executed one iteration of their loop and became blocked again at action \( bq_j \), \( 1 \leq j < N \). Indeed this happens because all these components have had only 1 empty packet to process. The exact order of execution in \( \xi_1 \) and \( t_{L2} \) is determined by priority assignment. At this point trace \( \rho \) can be expressed as:

\[ \rho = t_{\text{init}} \overset{1}{\sim} \tau_{\mathcal{C}_N} \sim t_{L1} \sim d(T_0^N + (l + 1) \ast T_N) \sim t_{\mathcal{C}_N}^2 \sim t_{L2} \sim u. \]

Because \( T_N < S < 2T_N \) at the end of \( t_{L2} \) there are no other actions ready than the delay action of \( C_N \). Therefore \( d(T_0^N + (l + 2) \ast T_N) \) is executed which advances time until moment \( T_0^N + (l + 2) \ast T_N \) is executed. Trace \( \rho \) can be expressed as:

\[ \rho = t_{\text{init}} \overset{1}{\sim} \tau_{\mathcal{C}_N} \sim t_{L1} \sim d(T_0^N + (l + 1) \ast T_N) \sim t_{\mathcal{C}_N}^2 \sim t_{L2} \sim d(T_0^N + (l + 2) \ast T_N) \sim u. \]

At this moment and state in the trace, the situation of components in the system is identical with the one at the end of \( t_{\text{init}} \), therefore the execution of the system repeats and can be expressed as:

\[ \rho = t_{\text{init}} \sim t_{\text{stable}} \]

\[ t_{\text{stable}} = (\exists \ t_{\mathcal{C}_N}^1 \sim t_l \sim d(T_0^N + i \ast T_N) \sim t_{\mathcal{C}_N}^2 \sim t_{L2} \sim d(T_0^N + i \ast T_N) \exists) \]

\[ \vdots \]

\[ \vdots \]
4.2 The interlaced standard

Also given the the repetitive beaviour of the system we also have for all \( i \in \mathbb{N} \) and \( s_1 \preceq fq_{N-1}^{i} \subseteq s_2 \preceq fq_N^{i} \subseteq s_3 \preceq bq_N^{i+1} \subseteq s_4 \preceq fq_N^{i+1} \subseteq t_{stable} \):

\[
\begin{align*}
\sigma(t_{init} \preceq s_2 \preceq fq_N^{i}) & < T_0^N + i * T_N + flush \land \\
\sigma(t_{init} \preceq s_1 \preceq fq_{N-1}^{i}) & - \delta(fq_{N-1}^{i}) \geq T_0^N + i * T_N \land \\
\sigma(t_{init} \preceq s_4 \preceq fq_N^{i+1}) & < T_0^N + (i+1) * T_N + flush \land \\
\sigma(t_{init} \preceq s_3 \preceq bq_N^{i+1}) & - \delta(bq_N^{i+1}) \geq T_0^N + (i+1) * T_N 
\end{align*}
\]

which implies that \( SQoSci(t_{stable}, t_{init}, fq_{N-1}, fq_N, bq_N, fq_N, T_0^N, T_N) \) holds.

B. When \( S < T_N \) the execution of \( t_{L} \) ends before moment \( T_0^N + (l+1) * T_N \) advances time until \( T_0^N + (l+1) * T_N \). Trace \( t_{L} \) is empty because in this case the cascaded execution of components in \( LSC_N \) is entirely recorded by \( t_{L} \). The rest of the statements are to be proved in the same manner as in case A.

As a direct consequence of the above theorem, the following corollary shows that during the stable phase, whenever \( C_N \) executes \( fq_{N-1} \), all the other components in the chain (\( C_i, 1 \leq i < N \)) are blocked at action \( bq \).

**Corollary 4.4.** Consider \( t \) a prefix of trace \( \rho \) such that \( t_{init} \preceq St_t(t) \) and the next action after \( t \) is \( bq_{N-1} \) if \( t_{init} \preceq t \preceq bq_{N-1} \). If \( S < 2T_N \) and \( P(C_N) = \max_{i=1}^{N} P(C_i) \) then \( C_i \ not \ bq \), for all \( i, 1 \leq i < N \).

**Proof.** Follows directly from Theorem 4.2.

\( P(C_N) = \max_{i=1}^{N} P(C_i) \) ensures that whenever \( C_N \) is \( ready \ to \ run \), it can run. This ensures \( C_N \) executes at a periodic rate as required by \( SQoSci \). We include below the corollaries showing how the QoS requirements can be satisfied during the finite prefix \( t_{init} \) and subsequently for the entire trace \( \rho \).

**Corollary 4.5.** \( S < 2T_N \land P(C_N) = \max_{i=1}^{N} P(C_i) \implies SQoSci(t_{init}, \varepsilon, fq_{N-1}, fq_N, bq_N, fq_N, T_0^N, T_N) \).

**Proof.** \( SQoSci(t_{init}, \varepsilon, fq_{N-1}, fq_N, bq_N, fq_N, T_0^N, T_N) \) holds when component \( C_N \) executes with rate \( T_N \) (within the flush time) during \( t_{init} \).

\( S < 2T_N \) implies that the rate at which packets are produced in \( fq_{N-1} \) is higher than their consumption rate. Therefore during \( t_{init} \), at the end of every \( 2T_N \) the length of \( fq_{N-1} \) is strictly positive. This implies that after the first execution of \( C_N \) during \( t_{init} \), \( C_N \) will never become blocked from channel perspective. It also means that \( C_N \) is ready to run from both time and channel perspective every \( T_N \) because according to its program, \( C_N \) needs to consume a full packet only once every \( 2T_N \).
When \( C_N \) has maximum priority, it also executes whenever it is ready to run which means that during \( t_{\text{init}} \), \( C_N \) will execute at rate \( T_N \) which implies that
\[
\text{SQoSci}(t_{\text{init}}, \varepsilon, f_{q_{N-1}}?, f_{q_N}!, b_{q_N}?, f_{q_N}!, T_0^N, T_N) \quad \text{holds.}
\]

**Corollary 4.6.** \( S < 2T_N \land P(C_N) = \max_{i=1}^N P(C_i) \Rightarrow \text{SQoSci}(\rho, \varepsilon, f_{q_{N-1}}?, f_{q_N}!, b_{q_N}?, f_{q_N}!, T_0^N, T_N) \).

**Proof.** Follows directly from Corollary 4.5 and Theorem 4.2.

When considering variable computation times for the actions of components by imposing \( SM < 2T_N \) where
\[
SM = \sum_{i=1}^N S_M(t_{C_i}).
\]
the stable phase is reached again and all lemmas and corollaries presented above hold as well, which means that again the trace \( \rho \) satisfies the \( QoS \) requirement. Indeed by ensuring at design time that \( SM < 2T_N \) where here \( SM \) takes into account the worst case computation time for all actions of components, the reasoning and the conclusions about the execution of the chain presented above hold again. In this case the above corollaries become:

**Corollary 4.7.** \( SM < 2T_N \land P(C_N) = \max_{i=1}^N P(C_i) \Rightarrow \text{SQoSci}(t_{\text{init}}, \varepsilon, f_{q_{N-1}}?, f_{q_N}!, b_{q_N}?, f_{q_N}!, T_0^N, T_N) \).

**Corollary 4.8.** \( SM < 2T_N \land P(C_N) = \max_{i=1}^N P(C_i) \Rightarrow \text{SQoSci}(\rho, \varepsilon, f_{q_{N-1}}?, f_{q_N}!, b_{q_N}?, f_{q_N}!, T_0^N, T_N) \).

Note that the \( SM < 2T_N \) restriction can be verified at the design of a chain by measuring the load on the processor of one iteration of each component.

### 4.2.3 Practical Applications

Trace \( \rho \) and its eager schedule can be calculated when knowing the computation times of all component actions, and by choosing in each state the ready action of the component with the highest priority. Knowing the trace and the schedule allows to calculate the number of context switches, and the number of actions needed to process a packet from input to the output of a chain. The schedule also renders the start and response times for individual tasks and the response time of the chain.

As in the previous case where \( C_N \) is a time-driven component, because \( SM < 2T_N \), component \( C_N \) has the same effect on the interleaved execution of the rest of components during the stable phase \( t_{\text{stable}} \) as a data-driven \( C_N \) with minimum priority (case described in Chapter 3). Due to this, we find again
that corollaries addressing the stable phase of a chain composed of only data-driven components (with \( C_N \) having the lowest priority) hold in this case as well. From here we deduce that:

- the minimum necessary and sufficient capacity of each queue in the chain is 1 (Corollary 3.3).
- the response time of the chain is to be calculated as shown in Chapter 3, in the case of a chain composed of only data driven components with \( m = N \). Also follows directly that response time of the chain cannot be improved by assigning the minimum priority to \( C_1 \) as suggested in Chapter 3 because that does not change the influence of the time-driven component \( C_N \).
- according to Theorem 3.3 the response time of the chain is reduced by reducing the capacities of queues preceding \( C_N \).
- the number of context switches occurring due to the interleaved execution of the data driven components during the stable phase can only be reduced by one context switch. This is the context switch due to preemption when \( C_N \) has a higher priority than other components in the chain. This context switch can be avoided by assigning to \( C_N \) the lowest priority in the chain. However this comes at the cost of a lower QoS when \( S > T_N \) given that in this situation predicate \( SQoSci(\rho, \varepsilon, fq_{N-1}, \text{true}, bq_N, \text{true}, \text{true}, b^N, T_0^N, T_N) \) is not satisfied.

4.2.4 Modeling overload situations

In the case of overload situations, for the same reasons as explained in section 4.1.4 we modify the constraint \( S < 2T_N \) to

\[
\sum_{i=k-M+1}^{k} S_i' + E \leq M \cdot 2 \cdot T_N, \ M \in \mathbb{N}.
\]

The QoS requirement in this case is specified as:

\[
SQoSci_{\text{overload}}(\rho) \overset{def}{=} (\exists pref, t, pref^\neg t = \rho : SQoSci(t, pref, f_{q_{N-1}}, \text{true}, bq_N, \text{true}, \text{true}, \text{true}, b^N, T_0^N, T_N)).
\]

By using a similar reasoning we find that Lemma 4.3 holds in this case as well. Also Lemma 4.4 becomes in this case
Lemma 4.6. Given that equation (4.12) is a property of the system execution, \( P(C_{N-1}) = \min_{i=1}^{N} P(C_i) \), \( P(C_N) = \max_{i=1}^{N} P(C_i) \), and \( \text{Cap}(f_{qN-1}) = M \), then there exist \( s_i \in St(\rho) \), \( i \in \mathbb{N} \) such that \( \sigma(s_i) = \sigma(t_{\text{init}}) + i \ast M \ast 2 \ast T_N \) and \( L(s_i, f_{qN-1}) = M \).

Proof. Identical approach as in the proof for Lemma 4.4. The only difference is that where we used \( T_N \) in the non-interlaced case becomes \( 2 \ast T_N \) in the interlaced case.

As in the non-interlaced case Corollary 4.9 regarding satisfying the QoS requirement follow immediately.

Corollary 4.9. Given that equation (4.12) characterizes the execution of the system, \( P(C_{N-1}) = \min_{i=1}^{N} P(C_i) \), \( P(C_N) = \max_{i=1}^{N} P(C_i) \), and \( \text{Cap}(f_{qN-1}) = M \), then \( SQoS_{\text{coreload}}(\rho) \) holds and hence \( SQoS_{\text{coverload}}(\rho) \) holds as well.

Proof. Similar proof as for Corollary 4.3.

4.3 A linear chain where the first component is time-driven

In this section we study the execution of a linear chain in which the first component is time driven (Figure 4.9). An example from practice of such a component is the video digitizer component. The video digitizer captures periodically video images (frames/fields) via a video camera. These images are passed on to the other components down the chain most commonly to be displayed on a TV screen, or to be encoded and saved on a storage facility. In the interlaced standard, the video digitizer captures an image (field) each period. The two fields are then combined into one frame that is eventually passed on in the chain. At the programming level, the basic statement \( \text{process}_{c} fch``(VAR, VAR) \) denotes the processing during the first period of the time-driven component.

Figure 4.9. Chain consisting of a time-driven component followed by data-driven components.
process\_f\_ct1^b(VAR,VAR) denotes the processing during the second period of the time-driven component. The program that specifying this behaviour is presented in Figure 4.10.

\[
C_1 : \{ \\
\text{getTime}(T_0^1); \; i := 0; \\
\text{while} \; \text{(true)} \; \text{do} \\
\{ \text{receive}(f_{q_0}, f_{Packet}, 1); \\
\text{// the first field of the image is stored in fPacket.} \\
\text{receive}(b_{q_1}, e_{Packet}, 1); \\
\text{process\_f\_ct1}(f_{Packet}, e_{Packet}); \\
\text{// first field is stored in ePacket.} \\
\text{send}(b_{q_0}, f_{Packet}, 1); \\
\}\; i := i + 1; \\
\text{delay\_until}(T_0^1 + i * T_1); \\
\text{receive}(f_{q_0}, f_{Packet}, 1); \\
\text{// the second field of the image is stored in fPacket.} \\
\text{process\_f\_ct1}(f_{Packet}, e_{Packet}); \\
\text{// two fields are merged into one image stored in ePacket.} \\
\text{send}(b_{q_0}, f_{Packet}, 1); \\
\text{send}(f_{q_1}, e_{Packet}, 1); \\
\text{// the image stored in ePacket is sent to C_2.} \\
\}\; i := i + 1; \\
\text{delay\_until}(T_0^1 + i * T_1); \} \\
\]

Figure 4.10. Program of a time-driven component \(C_1\). Implementation according to the interlaced standard.

The corresponding trace actions in the trace alphabet of the new basic statements are defined below:

\[
\text{Alph}'(\text{process\_f\_ct1}(\text{VAR LIST}')) \overset{def}{=} \{ c_1^{l'}, \ldots, c_1^{m_{l'}} \} \\
\text{Alph}'(\text{process\_f\_ct1}(\text{VAR LIST}')) \overset{def}{=} \{ c_1^{m_{l'}}, \ldots, c_1^{m_{l''}} \} 
\]

The trace associated with each basic statement is

\[
\text{Tr}'(\text{process\_f\_ct1}(\text{VAR LIST}')) \overset{def}{=} \exists c_1^{l'} \ldots \exists c_1^{m_{l'}} \\
\text{Tr}'(\text{process\_f\_ct1}(\text{VAR LIST}')) \overset{def}{=} \exists c_1^{m_{l'}}, \ldots, \exists c_1^{m_{l''}} 
\]

The associated trace set of component \(C_1\) is:
The meaning of the predicate above is that if
\[ SQoSci(\rho, \varepsilon, f_{q_0}^?, b_{q_0}!, f_{q_0}^?, f_{q_1}!, T_0^1, T_1) \] holds, then \( C_1 \) executes at a constant rate \( T_1 \), within the flush time interval, where:

\[
flush > \mu + \max(s_8^M(t_{C_1}^1) + s_8^M(t_{C_1}^2), s_8^M(t_{C_1}^1))
\]

(4.13)
4.3 A linear chain where the first component is time-driven

4.3.2 Characterization of system behaviour

We consider for the beginning fixed computation times for all component actions. We consider in this section that $S$ denotes the processing time between the production of two consecutive packets ($k$ and $k+1$) in $f_{q_i}$. $S$ is defined in (4.2). In the remainder of this section we show that $S < 2T_1$ and $P(C_1) = \max_{i=1}^{N} P(C_i)$ are sufficient conditions such that the QoS requirement would hold for the entire $\rho$. To prove this, we analyze first the behaviour of the chain when $S < 2T_1$ and $P(C_1) = \max_{i=1}^{N} P(C_i)$ are imposed.

**Theorem 4.3.** When $S < 2T_1$ and $P(C_1) = \max_{i=1}^{N} P(C_i)$ the pipeline system depicted in Figure 4.9 assumes a repetitive, periodical behavior called stable phase after a finite initial phase. The complete behavior is characterized by

$$\rho = t_{\text{init}} \sim t_{\text{stable}}$$

where

$$t_{\text{init}} = \hat{x} t^i_{C_1} \sim t^1_{C_1} \sim d(T^1_0 + i \ast T_1) \hat{x},$$

$$t_{\text{stable}} = (\hat{x} t^2_{C_1} \sim t_{R_1} \sim d(T^1_0 + i \ast T_1) \sim t^1_{C_1} \sim t_{R_2} \sim d(T^1_0 + i \ast T_1) \hat{x})^\omega$$

as illustrated in Figure 4.11 and:

- **Sub-traces** $t_{R_1}$ and $t_{R_2}$ record the interleaved execution of components in RSC$_1$.
- **Sub-trace** $t_{R_1}$ records the interleaved execution of components in RSC$_1$ between the execution end of $t^2_{C_1}$ and start of $d(T^1_0 + i \ast T_1)$.
- **Sub-trace** $t_{R_2}$ records the interleaved execution of components in RSC$_1$ between the execution end of $t^1_{C_1}$ and start of $d(T^1_0 + i \ast T_1)$ (of the next iteration) when $T_1 < S < 2T_1$. Trace $t_{R_2}$ is empty when $S < T_1$.
- **During one iteration of** $t_{\text{stable}}$ all components execute one iteration of their loop. At the end of each iteration of $t_{\text{stable}}$ components in RSC$_1$ are blocked at $f_{q_{i-1}}$?, $1 < i \leq N$.

**Proof.** We prove the statement of the theorem by construction of this trace $\rho$.

The assumption about the initial state of the queues in the chain (excepting $f_{q_0}$ and $b_{q_0}$) is that all forward queues are empty and all backward queues are full. This means that the initial state of all components except $C_1$ is that they are blocked at action $f_{q_{i-1}}$?.

The only component that is ready-to-run in the chain is $C_1$. Therefore trace
ρ starts with the first actions of $C_1$ according to its program (Figure 4.11):

$$t_{\text{init}} = \downarrow t_{C_1} ^{i} \sim t_{C_1} ^{i} \sim d(T_0 + T_1) \uparrow$$

Given that after action $bq_0!$ there are no other actions ready, action $d(T_0 + T_1)$ follows in the trace. Since $\sigma(t) < T_0 + T_1$ ($t_{\text{init}} = t \sim d(T_0 + T_1)$), hence action $d(T_0 + T_1)$ advances the time until moment $T_0 + T_1$, and the system experiences idle time.

At moment $T_0 + T_1$, $C_1$ becomes ready to run from time perspective, the other components in the chain are still blocked, therefore $C_1$ executes the following trace according to its program:

$$\downarrow fq_0? \sim c_1^{i} \sim \ldots \sim c_1^{m-1} \sim bq_0! \sim fq_1! \uparrow$$

When $fq_1!$ is executed, it de-blocks from channel perspective $C_2$. However given that $P(C_1) = \max_{i=1}^{N} P(C_i)$, $C_2$ cannot execute in this state, leaving $C_1$ to continue its execution with action $i := i + 1$ according to its program. This means that the sub-trace executed by $C_1$ after $t_{\text{init}}$ is

$$t_{C_1} ^{i} = \downarrow fq_0? \sim c_1^{i} \sim \ldots \sim c_1^{m-1} \sim bq_0! \sim fq_1! \sim i := i + 1$$

Also follows that $\rho$ can be expressed so far as:

$$\rho = t_{\text{init}} \sim t_{C_1} ^{i} \sim t$$

Given that in state $t_{\text{init}} \sim t_{C_1} ^{i}$ there exist other ready actions than $d(T_0 + 2 \times T_1)$ ($C_2$ has been de-blocked), implies that the delay action is postponed until all components in $RSC_1$ are blocked again, or until $\sigma(s \sim d(T_0 + 2 \times T_1)) > T_0 + 2 \times T_1$ where $t_{\text{init}} \sim t_{C_1} ^{i} \in St(s)$. 
Because $\sigma(t_{init} \sim t_{C_1}^1) < T_0^1 + 2 * T_1$, state $t_{init} \sim t_{C_1}^2$ is followed by actions of $C_2$. This component executes one iteration of its loop and becomes blocked again at action $fq_1^1$ because it has only one full packet to process. Moreover when $C_2$ executes $fq_2^1$, it de-blocks component $C_3$, which also executes one iteration of its loop and becomes blocked at action $fq_3^2$. In fact, in this manner all components in $RSC_1$ are de-blocked in cascade, execute one iteration of their loop and become blocked again at action $fq_{i-1}^i$, $1 < i \leq N$. The interleaved execution of components in $RSC_1$ is completely determined by the priority assignment. We denote the trace recording this execution with $\rho$. We distinguish the following case analysis:

A. When $T_1 < S < 2T_1$, the execution of components in $RSC_1$ is preempted at moment $T_0^1 + 2 * T_1 + \mu$ by $C_1$ which has the highest priority in the chain. $C_1$ executes $d(T_0^1 + 2 * T_1)$ because $\sigma(s - d(T_0^1 + 2 * T_1)) \geq T_0^1 + 2 * T_1$. The interleaved execution of components in $RSC_1$ between $t_{C_1}^2$ and $d(T_0^1 + 2 * T_1)$ is denoted with $t_{R_1}$.

After $d(T_0^1 + 2 * T_1)$, $C_1$ executes $t_{C_1}^1$, according to its program. At this point $\rho$ can be expressed as:

$$\rho = t_{init} \sim t_{C_1}^2 \sim t_{R_1} \sim d(T_0^1 + 2 * T_1) \sim t_{C_1}^1 \sim t_1, \text{ where}$$

$$t_{C_1}^1 = \tau (true \simfqq \sim bq_1 \sim c_1^1 \sim \ldots \sim c_{1}^{m_{1}} \sim bq_0) \sim i := i + 1$$

Sub-trace $t_{C_1}^1$ is followed by the rest of the actions in $t_{R_1}$, denoted as $t_{R_2}$. $\rho$ can be expressed as:

$$\rho = t_{init} \sim t_{C_1}^2 \sim t_{R_1} \sim d(T_0^1 + 2 * T_1) \sim t_{C_1}^1 \sim t_{R_2} \sim t_2.$$  

At the end of $t_{R_2}$, all components in $RSC_1$ are blocked at action $fq_{i-1}^i$, $1 < i \leq N$. Therefore the action following $t_{R_2}$ is $d(T_0^1 + 3 * T_1)$.

$S < 2T_1$ implies $\sigma(t_{init} \sim t_{C_1}^2 \sim t_{R_1} \sim d(T_0^1 + 2 * T_1) \sim t_{C_1}^1 \sim t_{R_2}) < T_0^1 + 3 * T_1$. This means that $d(T_0^1 + 3 * T_1)$ advances time until $T_0^1 + 3 * T_1$. $\rho$ can be expressed as:

$$\rho = t_{init} \sim t_{C_1}^2 \sim t_{R_1} \sim d(T_0^1 + 2 * T_1) \sim t_{C_1}^1 \sim t_{R_2} \sim d(T_0^1 + 3 * T_1) \sim t_3.$$  

At moment $T_0^1 + 3 * T_1$ component $C_1$ becomes ready from time perspective while the other components are all blocked. This situation is identical with the one at the end of $t_{init}$ which implies that the execution of the system will repeat in the same manner as presented above.

B. When $S < T_1$, the execution of $t_{R_1}$ records the entire cascaded execution of components in $RSC_1$, therefore $t_{R_1}$ is empty. $t_{R_1}$ ends before moment $T_0^1 + 2 * T_1$ and $d(T_0^1 + 2 * T_1)$ advances time until $T_0^1 + 2 * T_1$. 

$\square$
The following corollary states that when $S < T_1$, whenever $C_1$ executes $fq_0? (p_1)$, all the other components in the chain ($C_i, 2 \leq i \leq N$) are blocked at action $fq_{i-1}$?

**Corollary 4.10.** Consider $t$ a prefix of trace $\rho$ such that $t_{\text{init}} \in St(t)$ and the next action after $t$ is $fq_1! (t_{\text{init}} \subseteq t \Rightarrowfq_1! \subseteq \rho)$. If $S < T_1$ and $P(C_1) = \max_{i=1}^{N} P(C_i)$ then $C_i b \ f_{q_{i-1}}$. for all $i, 1 \leq i < N$.

**Proof.** Follows directly from Theorem 4.3.

**Corollary 4.11.** $S < 2T_1 \land P(C_1) = \max_{i=1}^{N} P(C_i) \Rightarrow S\text{QoS}c(i, e, fq_0?, bq_0!, fq_0?, f_1, T_0^1, T_1)$.

**Proof.** Follows directly from Theorem 4.3 (Figure 4.11).

When considering variable computation times for the actions of components by imposing $S^M < 2T_1$ where

$$S^M = \sum_{i=1}^{N} S^M (t_C).$$

the stable phase is reached again and all lemmas and corollaries presented above hold as well, which means that again the trace $\rho$ satisfies the QoS requirement. Indeed by ensuring at design time that $S^M < 2T_1$ where here $S^M$ takes into account the worst case computation time for all actions of components, the reasoning and the conclusions about the execution of the chain presented above hold again. Such a restriction in the design of a chain can be verified by measuring the load on the processor of one iteration of each component.

### 4.3.3 Practical Applications

Trace $\rho$ and its eager schedule can be calculated when knowing the computation times of all component actions, and by choosing in each state the ready action of the component with the highest priority. Knowing the trace and the schedule allows to calculate the number of context switches, and the number of actions needed to process a packet from input to the output of a chain. The schedule also renders the start and response times for individual tasks and the response time of the chain.

Important to note is that because $S^M < 2T_1$, regardless of its priority, component $C_1$ has the same effect on the interleaved execution of the rest of components during the stable phase as a data-driven $C_1$ with minimum priority (as described in Chapter 3). Owing to this fact, we find that corollaries addressing
4.4 A video surveillance system

In this section we present the analysis of the execution of a surveillance system. The system consists of a video digitizer component as presented in section 4.3, a video renderer component (section 4.2) and a number of data-driven components. The video digitizer and the video renderer execute according to the interlaced standard. The data-driven components have the role of improving through additional processing the video frames received from the video digitizer.

For the purposes of this analysis we consider the periods of the video digitizer and of the video renderer equal:

\[ T_1 = T_N \] (4.14)
Figure 4.12. Surveillance system consisting of $N - 2$ data-driven components and a time-driven components at the beginning and respectively end extremity.

We also assume that $C_1$ has the highest priority in the chain, and $C_N$ the second highest priority.

$$P(C_1) = \max_{i=1}^{N} P(C_i) \land P(C_N) = \max_{i=2}^{N} P(C_i)$$

(4.15)

The QoS requirement is as expected a combination of the QoS requirements as defined in section 4.2 and section 4.3.1:

$$S\text{QoS}(t, \text{pref}) \overset{\text{def}}{=} S\text{QoS}(t_1, \text{pref}, f_{q_0}?, b_{q_0}!, f_{q_1}?, f_{q_1}!, T_0^1, T_1) \land S\text{QoS}(t, \text{pref}, f_{q_{N-1}}?, f_{q_{N-1}}!, b_{q_{N-1}}?, f_{q_{N-1}}!, T_0^N, T_N).$$

We consider variable computation times for the components actions and

$$S^M = \sum_{i=1}^{N} S^M(t_C).$$

(4.16)

We denote with MSC the sub-chain consisting of components $C_i, 2 \leq i \leq N - 1$. We denote with $\Delta_k$ the time between the execution start of $t^k_C$ during which a packet $k$ is produced in $f_{q_1}$, until the beginning of the execution of $t^k_{CN}$ (for $k > 1$), or $t^k_{CN} - t^k_{CN}(k = 1)$, during which packet $k$ is consumed from $f_{q_{N-1}}$:

$$\Delta_k = \begin{cases} \sigma(u^1_{CN}) - S_{\delta}(t^1_{CN}) - S_{\delta}(t^2_{CN}) - S_{\delta}(t^2_{CN}) & k = 1 \\ \sigma(u^1_{CN}) - S_{\delta}(t^1_{CN}) - \sigma(v^2_{CN}) - S_{\delta}(t^2_{CN}) & k > 1. \end{cases}$$

Theorem 4.4. Consider a system as in Figure 4.12. Given that (4.14), (4.15) and $S^M < \Delta_1 < 2T_1$ hold, the video surveillance system assumes a repetitive behavior is characterized by

$$\rho = t_{\text{init}} \sim t_{\text{stable}}$$
Figure 4.13. Initial and stable phase of a video surveillance system.

\[ t_{\text{init}} = \times t^1_{C_1} - t^1_{C_1} - d(T_0^1 + T_1) - t^2_{C_1} - t^1_{MSC} - d(T_0^1 + 2 * T_1) - t^2_{C_1} - t^2_{MSC} - t^3_{C_N} - t^1_{C_N} - d(T_0^1 + 3 * T_1) \]

\[ t_{\text{stable}} = (\times t^2_{C_1} - t^3_{MSC} - d(T_0^N + i * T_N) - t^2_{C_N} - t^4_{MSC} - d(T_0^1 + i * T_1) - t^1_{C_1} - t^5_{MSC} - d(T_0^N + i * T_N) - t^1_{C_N} - d(T_0^1 + i * T_1))^{\infty} \]

as illustrated in Figure 4.13. Also:

- Sub-traces \( t^1_{MSC}, t^2_{MSC}, t^3_{MSC}, t^4_{MSC}, t^5_{MSC} \) record interleaved executions of components in sub-chain MSC.
- Sub-traces \( t^2_{MSC} \) and \( t^5_{MSC} \) are empty when \( S^M < T_1 \).
- During one iteration of \( t_{\text{stable}} \) all components execute one iteration of their loop. At the end of each iteration of \( t_{\text{stable}} \) components in MSC are blocked at \( f_{qi^-1} \), \( 1 < i \leq N - 1 \).
- \( \Delta_k = T_0^N - T_0^1 - T_1, \ \forall k \in \mathbb{N}, \ k > 0 \).
- \( SQoSc(t_{\text{stable}}, t_{\text{init}}) \) holds.

**Proof.** We prove the statements of the theorem by construction of this trace \( p \).

**I.** We start by proving that the execution and properties of the system during \( t_{\text{init}} \) are as stated in the theorem.

The assumption about the initial state of the queues in the chain (excepting \( f_{q_0} \) and \( b_{q_0} \)) is that all forward queues are empty and all backward queues are full. This means that the initial state of all components except \( C_1 \) is that they are blocked at action \( f_{q_{i-1}} \). The only component that can execute is \( C_1 \). Component \( C_1 \) starts at time \( T_0^1 \). Given that the execution start of \( C_1 \) marks also the execution start of the system, we consider
According to the program of $C_1$, $t^1_{C_1} \sim t^1_{C_1}$ are executed. Since in state $t^1_{C_1} \sim t^1_{C_1}$ all other components are still blocked, the delay action $d(T^1_0 + T_1)$ of $C_1$ can be executed. Therefore at this point $\rho$ can be expressed as:

$$\rho = t^1_{C_1} \sim t^1_{C_1} \sim d(T^1_0 + T_1) \sim u.$$ 

Action $d(T^1_0 + T_1)$ advances time until moment $T^1_0 + T_1$. At this moment the only component ready-to-run is still $C_1$. This implies that the execution of the system continues with $t^2_{C_1}$ according to the program of $C_1$. $\rho$ can be expressed as:

$$\rho = t^1_{C_1} \sim t^1_{C_1} \sim d(T^1_0 + T_1) \sim t^2_{C_1} \sim u.$$ 

During $t^2_{C_1}$, action $fq_1!$ is executed which makes component $C_2$ to become ready-to-run from channel perspective. This postpones the execution of $d(T^1_0 + 2 \ast T_1)$ according to the conditions expressed by predicate $Ssc$. Instead $t^2_{C_1}$ is followed by an execution in cascade of components in $MSC$. That is because when action $fq_1!$ is executed during $t^2_{C_1}$, all components in $MSC$ are de-blocked in cascade, execute one iteration of their loop and become blocked again at action $fq_{i-1}?$, $1 < i \leq N - 1$.

**A.** When $T_1 < S^M < 2T_1$, the interleaved execution of components in $MSC$ continues until moment $T^1_0 + 2 \ast T_1 + \mu$ when it is preempted (on an atomic action boundary) by action $d(T^1_0 + 2 \ast T_1)$ of $C_1$ ($P(C_1) = \max_{i=1}^{N} P(C_i)$). The interleaved execution of components in $MSC$ until moment $T^1_0 + 2 \ast T_1 + \mu$ is recorded by trace $t^1_{MSC}$. The delay action $d(T^1_0 + 2 \ast T_1)$ is followed by $t^1_{C_1}$ according to the program of $C_1$. So far $\rho$ can be expressed as:

$$\rho = t^1_{C_1} \sim t^1_{C_1} \sim d(T^1_0 + T_1) \sim t^2_{C_1} \sim t^1_{MSC} \sim d(T^1_0 + 2 \ast T_1) \sim t^1_{C_1} \sim u.$$ 

After $t^1_{C_1}$, $C_1$ becomes blocked from time perspective and the interleaved execution of components in $MSC$ is resumed. The delay action $d(T^1_0 + 3 \ast T_1)$ is postponed because there exists a ready action of a component in $MSC$ other than delay. This second part of the interleaved execution of components in $MSC$ is denoted by $t^2_{MSC}$.

At the end of $t^2_{MSC}$, action $fq_{N-1}!$ is executed. This de-blocks component $C_N$ at action $gr(T^N_0)$. Given the fact that $C_1$ is blocked, and $P(C_N) = \max_{i=2}^{N} P(C_i)$, trace $t^c_{C_N} \sim t^1_{C_N}$ of $C_N$ are executed according to the program of $C_N$. $\rho$ can be expressed as:

$$\rho = t^1_{C_1} \sim t^1_{C_1} \sim d(T^1_0 + T_1) \sim t^2_{C_1} \sim t^1_{MSC} \sim d(T^1_0 + 2 \ast T_1) \sim t^1_{C_1} \sim t^2_{MSC} \sim t^c_{C_N} \sim t^1_{C_N} \sim u.$$
After $t^1_{CN}$, $CN$ becomes blocked from time perspective. Indeed, although there are no other regular action ready in this state, there exists another delay action \( d(T^1_0 + 3 \times T_1) \) for which the delay time is due earlier \( T^1_0 + 3 \times T_1 < T^N_0 + T_N \). Therefore after $t^1_{CN}$, according to $S\sigma_c$, $C_1$ becomes ready from time perspective and since no other components are ready-to-run in this state, $d(T^1_0 + 3 \times T_1)$ is executed. Trace $\rho$ becomes:

\[
\rho = 3t^2_{C_1} \sim t^1_{C_1} \sim d(T^1_0 + T_1) \sim t^2_{C_1} \sim t^1_{MSC} \sim d(T^1_0 + 2 \times T_1) \sim t^1_{C_1} \sim t^2_{MSC} \sim t^1_{CN} \sim d(T^1_0 + 3 \times T_1) \sim u^1.
\]

We have therefore so far that $\rho = t_{init} \sim u$.

**B.** When $S^M < T_1, t^1_{MSC}$ records the entire interleaved execution of components in $MSC$ and therefore $t^2_{MSC}$ is empty. The rest of the actions sequence in $t_{init}$ remains the same for the same reasons as explained at A.

So far, from the system execution (illustrated in Figure 4.13) we conclude that during $t_{init}$:

- $t^1_{C_1} \sim t^1_{C_1}$ starts executing at moment $T^1_0$.
- $t^2_{C_1}$ starts executing at moment $T^1_0 + T_1$.
- $t^1_{C_1}$ starts executing at moment $T^1_0 + 2 \times T_1$.
- $t^2_{CN}$ starts executing at moment $T^N_0$.

Given (4.13) and (4.7) follows directly that $SQoSci(t_{init}, \varepsilon, f_{q_0}^1, b_{q_0}, f_{q_1}^1, f_{q_0}^2, T^1_0, T_1)$ and $SQoSci(t_{init}, \varepsilon, f_{q_{N-1}}^N, f_{q_N}, b_{q_N}, f_{q_N}^N, T^N_0, T_N)$ hold, which implies that $SQoSc(t_{init}, \varepsilon)$ holds. Also results immediately that

\[
\Delta_1 = T^N_0 - T^1_0 - T_1. \quad (4.17)
\]

**II.** We continue with proving that after $t_{init}$ the execution of the system becomes repetitive. Since $S^M < \Delta_1 < 2T_1$ follows that $\sigma(t \sim t^1_{CN}) < T^1_0 + 3 \times T_1$. Therefore the last action in $t_{init}$, $d(T^1_0 + 3 \times T_1)$ advances time until moment $T^1_0 + 3 \times T_1$.

At moment $T^1_0 + 3 \times T_1$, $C_1$ continues its execution according to its program with $t^2_{C_1}$. This means that

\[
\sigma(t \sim t^2_{C_1}) - S_8(t^2_{C_1}) = T^1_0 + 3 \times T_1 \quad (4.18)
\]

Trace $\rho$ can be expressed at this point as:

\[
\rho = t_{init} \sim t^2_{C_1} \sim u.
\]
During $t^2_{C_1}$ action $fq_1$ executed which determines $C_2$ to become ready-to-run from channel perspective. Since in this state there exists another action than a delay that is ready, $d(T^0_0 + 4* T_1)$ is postponed. $t^2_{C_1}$ is followed by actions of $C_2$. $C_2$ executes one iteration of its loop and becomes blocked again at action $fq_1$ because it has only one full packet to process. When $C_2$ executes $fq_2$, it causes a de-blocking in cascade of components in $MSC$. All components in $MSC$ execute one iteration of their loop and become blocked again at action $fq_{i-1}$?, $1 < i \leq N - 1$.

C. When $T_1 < s^M < 2T_1$, the interleaved execution of components in $MSC$ continues until moment $T^N_0 + T_N + \mu$ when $C_N$ becomes ready-to-run from time perspective. Because $P(C_N) = \max_{i=2}^N P(C_i)$, $C_N$ preempts this interleaved execution by executing action $d(T^N_0 + T_N)$. The interleaved execution of components in $MSC$ until moment $T^N_0 + T_N + \mu$ is recorded by trace $t^3_{MSC}$. Trace $\rho$ can be expressed at this point as:

$$\rho = t_{init} - t_{C_1}^2 - t^3_{MSC} - d(T^N_0 + T_N) - u.$$  

At moment $T^N_0 + T_N + \mu$ the delay action $d(T^N_0 + T_N)$ does not have any effect. The execution continues with $t^2_{CN}$ according to the program of $C_N$. That is because at moment $T^N_0 + T_N$, $C_1$ is still blocked from time perspective and $C_N$ has a higher priority than all components in $MSC$. This means that

$$\sigma(t - t_{C_N}^2) - S(t_{C_N}^2) = T^N_0 + T_N + \mu \tag{4.19}$$

After $t^2_{CN}$, the interleaved execution of components in $MSC$ is resumed and the delay action $d(T^N_0 + 2* T_N)$ of $C_N$ is postponed given that in the current state there exist ready actions other than delay.

The interleaved execution of components in $MSC$ continues until moment $T^1_0 + 4* T_1 + \mu$ when $C_1$ becomes ready-to-run from time perspective. Because $P(C_1) = \max_{i=1}^N P(C_i)$, $C_1$ preempts this interleaved execution by executing action $d(T^1_0 + 4* T_N)$. The interleaved execution of components in $MSC$ until moment $T^1_0 + 4* T_1 + \mu$ is recorded by trace $t^4_{MSC}$. Trace $\rho$ can be expressed at this point as:

$$\rho = t_{init} - t_{C_1}^2 - t^3_{MSC} - d(T^N_0 + T_N) - t_{C_N}^2 - t^4_{MSC} - d(T^1_0 + 4* T_1) - u.$$  

At moment $T^1_0 + 4* T_1$ the delay action $d(T^1_0 + 4* T_1)$ does not have any effect of advancing time. The execution continues with $t_{C_1}^1$ given that $C_1$ has the highest priority in the chain. This means that

$$\sigma(t - t_{C_1}^1) - S(t_{C_1}^1) = T^1_0 + 4* T_1 + \mu \tag{4.20}$$

At the end of $t_{C_1}^1$ the interleaved execution of components in $MSC$ is re-
4.4 A video surveillance system

sumed and the delay action \( d(T_0 + 5 \cdot T_1) \) is postponed. We show below that the interleaved execution ends before moment \( T_0^N + 2 \cdot T_N \). The sub-trace that records the last part of the interleaved execution of components in \( MSC \) is \( s_{MSC} \).

Hence we will show that \( \sigma(t \sim t_{MSC}^5) < T_0^N + 2 \cdot T_N \).

Given the system execution so far we have:

\[
\sigma(t \sim t_{MSC}^5) = \\
\sigma(t_{init}) + S_6(t_{C_1}^2) + S_6(t_{MSC}^3) + S_6(t_{C_n}^2) + S_6(t_{MSC}^4) + \\
S_6(t_{C_1}^1) + S_6(t_{MSC}^5).
\]

\[
\begin{align*}
&= T_0^1 + 3 \cdot T_1 + S_6(t_{C_1}^2) + S_6(t_{MSC}^3) + S_6(t_{C_n}^2) + S_6(t_{MSC}^4) + \\
&\quad S_6(t_{C_1}^1) + S_6(t_{MSC}^5) < (4.16) \\
&< T_0^1 + 3 \cdot T_1 + S_M
\end{align*}
\]

\[
\begin{align*}
&= T_0^1 + 3 \cdot T_1 + T_0^N - T_0^1 - T_1
\end{align*}
\]

\[
\begin{align*}
&= (4.14) \\
&= T_0^N + 2 \cdot T_1
\end{align*}
\]

Hence we have that

\( \sigma(t \sim t_{MSC}^5) < T_0^N + 2 \cdot T_N \).

Trace \( \rho \) can be expressed at this point as:

\[
\rho = t_{init} \sim t_{C_1}^2 \sim t_{MSC}^3 \sim d(T_0^N + T_N) \sim t_{C_n}^2 \sim t_{MSC}^4 \sim d(T_0^1 + 4 \cdot T_1) \sim \\
\sim t_{C_1}^1 \sim t_{MSC}^5 \sim \mu \cdot.
\]

At the end of \( t_{MSC}^5 \) all components in \( MSC \) have executed one iteration of their loop and have become blocked again at action \( f_{q-1}^? \), \( 2 < i < N \). Given that at this point no regular actions are ready, the delay action \( d(T_0^N + 2 \cdot T_N) \) of \( C_N \) is executed. From (4.17), (4.14) and \( \Delta_i < 2T_1 \) follows directly that \( T_0^N + 2 \cdot T_N < T_0^1 + 5 \cdot T_1 \). Hence, according to \( S_{Sc} \), \( d(T_0^N + 2 \cdot T_N) \) is executed before \( d(T_0^1 + 5 \cdot T_1) \).

The delay action \( d(T_0^N + 2 \cdot T_N) \) advances time until \( T_0^N + 2 \cdot T_N \) when the execution of \( C_N \) continues according to its program with \( t_{C_n}^1 \). This means that

\[
\sigma(t \sim t_{C_n}^1) - S_8(t_{C_n}^1) = T_0^N + 2 \cdot T_N \quad (4.21)
\]

We have therefore that
\[\Delta_2\]
\[= T_0^N + 2 \cdot T_N - (T_0^1 + 3 \cdot T_1)\]
\[(4.14)\]
\[= T_0^N + 2 \cdot T_1 - (T_0^1 + 3 \cdot T_1)\]
\[= T_0^N - T_0^1 - T_1.\]

Because \(S_M < 2T_1\), \(t_{C_n}^1\) ends before moment \(T_0^1 + 5 \cdot T_1\). After \(t_{C_n}^1\) the delay action \(d(T_0^N + 2 \cdot T_N)\) is postponed because \(T_0^N + 2 \cdot T_N < T_0^1 + 5 \cdot T_1\). All components in MSC are still blocked, therefore the next action is \(d(T_0^1 + 5 \cdot T_1)\). The delay action \(d(T_0^1 + 5 \cdot T_1)\) advances time until moment \(T_0^1 + 5 \cdot T_1\). Trace \(\rho\) can be expressed at this point as:

\[\rho = t_{init} \searrow 3t_{C_1}^2 \searrow t_{MSC}^3 \searrow d(T_0^N + T_N) \searrow t_{C_n}^2 \searrow t_{MSC}^4 \searrow d(T_0^1 + 4 \cdot T_1) \searrow t_{C_1}^1 \searrow t_{MSC}^5 \searrow d(T_0^N + 2 \cdot T_N) \searrow t_{C_n}^1 \searrow d(T_0^1 + 5 \cdot T_1) \searrow u\).

At moment \(T_0^1 + 5 \cdot T_1\) all component in MSC are blocked at action \(f_{q_{i-1}}\), \(1 < i < N\), \(C_N\) is blocked from time perspective and \(C_1\) is ready from time perspective. That means that the state of all components in the chain is exactly the same as at moment \(T_0^1 + 3 \cdot T_1\) which marks the end of the \(t_{init}\). This means that the execution of the system is repetitive and trace \(\rho\) can be expressed as:

\[\rho = t_{init} \searrow 3t_{C_1}^2 \searrow t_{MSC}^3 \searrow d(T_0^N + T_N) \searrow t_{C_n}^2 \searrow t_{MSC}^4 \searrow d(T_0^1 + 4 \cdot T_1) \searrow t_{C_1}^1 \searrow t_{MSC}^5 \searrow d(T_0^N + 2 \cdot T_N) \searrow t_{C_n}^1 \searrow d(T_0^1 + 5 \cdot T_1) \searrow u\).

or as:

\[\rho = t_{init} \searrow t_{stable}.\]

D. When \(S_M < T_1\), \(t_{MSC}^3\) and \(t_{MSC}^4\) record the entire interleaved execution of components in MSC and for this reason \(t_{MSC}^5\) is empty. The rest of the actions sequence in \(t_{stable}\) remains the same for the same reasons as explained at C.

So far, from the system execution (illustrated in Figure 4.13) we conclude that during the first iteration of \(t_{stable}\):

- \(t_{C_1}^1\) starts as of moment \(T_0^1 + 3 \cdot T_1\). (4.18)
- \(t_{C_1}^1\) starts as of moment \(T_0^1 + 4 \cdot T_1 + \mu\). (4.20)
- \(t_{C_n}^2\) starts as of moment \(T_0^N + T_N + \mu\). (4.19)
- \(t_{C_n}^1\) starts as of moment \(T_0^N + 2 \cdot T_N\). (4.21)
Given the repetitive execution of the system during $t_{\text{stable}}$ follows that during each iteration of this sub-trace of $\rho$:

- $t_{C_1}^2$ starts as of each moment $T_0^1 + k \ast T_1, \forall k \in \mathbb{N}, k > 2$.
- $t_{C_1}^1$ starts as of each moment $T_0^1 + (k + 1) \ast T_1 + \mu, \forall k \in \mathbb{N}, k > 2$.
- $t_{C_N}^2$ starts as of each moment $T_0^N + k \ast T_N + \mu, \forall k \in \mathbb{N}, k > 1$.
- $t_{C_N}^1$ starts as of each moment $T_0^N + (k + 1) \ast T_N, \forall k \in \mathbb{N}, k > 1$.

Given (4.13) and (4.7) follows directly that $SQoSci(t_{\text{stable}}, t_{\text{init}}, f_{q_0}^0, b_{q_0}^0, f_{q_1}^1, T_0^1, T_1)$ and $SQoSci(t_{\text{stable}}, t_{\text{init}}, f_{q_{N-1}}^0, f_{q_N}^1, b_{q_N}^0, f_{q_N}^1, T_0^N, T_N)$ hold, which implies that $SQoSc(t_{\text{stable}}, t_{\text{init}})$ holds. Since we also have shown that $SQoSc(t_{\text{init}}, \varepsilon)$ holds, results directly that $SQoSc(\rho, \varepsilon)$ holds.

Due to the repetitive behaviour we also find that

$$\Delta_k = T_0^N - T_0^1 - T_1^N, \forall k \in \mathbb{N}, k > 0.$$ 

\[\square\]

**Corollary 4.12.** $S^M < \Delta_1 < 2T_1 \land ((P(C_1) = \max_{i=1}^{N} P(C_i) \land P(C_N) = \max_{i=2}^{N} P(C_i)) \land T_1 = T_N \Rightarrow \forall s \neg f_{q_1}^1 \subseteq \rho, C_i \notin f_{q_{i-1}}^i \text{ [in } s \text{ of } T_{cc}] (1 < i < N).$

**Proof.** Follows directly from Theorem 4.4. \[\square\]

**Corollary 4.13.** $S^M < \Delta_1 < 2T_1 \land ((P(C_1) = \max_{i=1}^{N} P(C_i) \land P(C_N) = \max_{i=2}^{N} P(C_i)) \land T_1 = T_N \Rightarrow SQoSc(\rho, \varepsilon).$

**Proof.** Follows directly from Theorem 4.4. \[\square\]

In practice, the conditions suggested in Theorem 4.4 can be imposed by measuring the duration of $S^M$ and ensuring at design time that $S^M < 2T_1$, and controlling the start of the first iteration of $C_N$ so that $S^M < \Delta_1 < 2T_1$. This can be done by adding a delay action at the startup of $C_N$ such that $C_N$ does not start as soon as it becomes ready to run from channel perspective but no sooner than $T_0^1 + T_1 + S^M$. In this case $t_{C_N}^N$ would become:

$$t_{C_N}^e = \varepsilon d(T_0^1 + T_1 + S^M) - gt(T_0^N) - i := 0 \varepsilon$$

Condition $S^M < 2T_1$ is reasonable to accommodate in practice because variations in computation times of components in the chain are not large. Video upscaling is excluded given that we consider equal periods for the video digitizer and the video renderer. For this reason we do not approach overload situations in this section.
4.4.1 Practical Applications

Given the fact that when \( S^M < \Delta_1 < 2T_1 \), \( P(C_1) = \max_{i=1}^{N} P(C_i) \), \( P(C_N) = \max_{i=2}^{N} P(C_i) \), and \( T_1 = T_N \), all components in \( RSC_1 \) depend on \( C_1 \), the reasoning regarding practical applications is similar to that used in section 4.3.3. Hence we find that:

- the minimum necessary and sufficient capacity of each queue in the chain is 1 (Corollary 3.3)
- the response time of the chain is to be calculated as shown in Chapter 3, in the case of a chain composed of only data-driven components with \( m = 1 \). The response time is not improved by assigning the minimum priority to \( C_1 \) because time-driven \( C_1 \) already has the effect of a component with minimum priority on the execution of the data-driven components in terms of the queue filling.
- the number of context switches occurring due to the interleaved execution of the data-driven components during the stable phase is not improved more by assigning the minimum priority to \( C_1 \) as suggested in Theorem 3.2 b-(i). That would be superfluous given that the time-driven component \( C_1 \) has the same effect as a data-driven \( C_1 \) with lowest priority in the chain. NCS is improved in this situation by taking \( P(C_2) > P(C_3) > \ldots > P(C_{N-1}) \) as suggested by the same theorem.

4.5 Summary

In this chapter we have presented a behavioural analysis of four types of systems with timing constraints. In all cases components can have variable computation times. data-driven components and ending with a time-driven component (section 4.1). Section 4.2 presents the behavioral analysis in the case where the last component is time-driven and executes according to the interleaving standard. The case of a system where the first component is time-driven executing according to the interleaving standard and the rest of the \( N-1 \) components are data-driven is presented in section 4.3. Finally, section 4.4 we show the behavioural analysis of a system encountered in practice. This is the case of video-surveillance system where the first and last components are time-driven and the rest of \( N-2 \) components are data-driven.

In each of the cases presented in sections 4.1, 4.2, 4.3 and 4.4 we specify the behavior of the chain by means of a trace \( \rho \) of the actions of the components that make up the chain, and the associated schedule function. We have proven that when overload situations are prevented at chain design time, af-
4.5 Summary

After a finite prefix (initial phase) the trace recording the execution of the chain becomes repetitive and periodic (stable phase). We have also shown that the chain satisfies the quality of service requirements for the entire trace $\rho$. In addition, in the first three systems analyzed we have explained that the time-driven component has the same influence on the overall execution of this chain as a data-driven component with minimum priority has on a chain composed of only data-driven components. This reduces the analysis of this time-driven system to be identical to that of the data-driven system in Chapter 3. As a result issues as the calculation and optimization of the necessary and sufficient memory in each buffer, the response time of the chain and the number of context switches occurring during the chain execution are to be reasoned about in the same way.

In the case of a system that contains two time-driven components, we show how the chain can be designed so that the execution of the system is driven by only one of the time-driven components namely the first one, while the quality of service requirements of both time-driven components are satisfied.

In the case where CPU overload situations are allowed at the design time of the chain (as a tradeoff for less processing speed and more components in the chain), we show that there does exist an infinite suffix of the trace $\rho$ during which the chain satisfies the quality of service requirements, provided that the capacity of the queue which connect the time-driven component to the rest of the chain is of a certain minimum. We calculate this minimum necessary and sufficient capacity. The results of this analysis are very relevant because it shows how to trade memory for lower processing power when designing systems that experience high variations in computation times of tasks. The trade-off proposed is very advantageous because the cost of additional amount of memory is much lower that the cost of processing power when CPU is overprovisioned to accommodate computational peaks.
In this chapter we introduce a new type of components called *components with deferred execution*. A component with deferred execution corresponds to a task that delegates part of the processing to some other hardware. Examples of components with deferred execution in TSSA are the *file reader* and the *file writer* component which retrieve and respectively store the media stream from/on the hard disk or a DVD disk.

We start this chapter by presenting the program and associated trace set of a component with deferred execution in section 5.1. In essence the analysis of this chapter shows what is the influence on the overall system execution of adding a component with deferred execution to the systems previously studied in Chapter 3 and Chapter 4. As such, as a first step we analyze in section 5.2 the execution of a linear chain where the first component is with deferred execution and the rest of the components are data-driven.

In section 5.3 we continue by adding a time-driven component at the end of the chain we studied in section 5.2. The new system is analyzed again to observe the combined influence of a component with deferred execution with the timing constraints induced by the time-driven component, on the overall chain execution.
The last section of this chapter presents the analysis of the mirrored case, where the chain is composed of a time-driven component, \( N - 2 \) data-driven components and ends with a component with deferred execution.

In all cases analyzed we observe again the repetitive nature of the system execution and we analyze the influence of the component with deferred execution on the overall execution of the system. Practical applications regard again the optimization of memory, number of context switches and response time. When comparing with the practical applications presented in the previous chapter, a distinguishing issue tackled here is the optimization of CPU utilization by eliminating the potential idle times occurring during the deferral times of the component with deferred execution. In the sections where systems with timing constraints are analyzed (section 5.3 and section 5.4), we use a similar approach as presented in Chapter 4 to show that the system execution is driven by the execution of the time-driven component. In all systems with timing constraints we study in this chapter, we address quality of service concerns as well.

### 5.1 A linear chain where the first component is with deferred execution

In this section we analyze the behavior of a linear chain composed of a component with deferred execution and \( N - 1 \) data-driven components as shown in Figure 5.1. A program example of a component with deferred execution at the beginning of the chain is shown in Figure 5.2. Note here that the `delay` actions in the program model the effect of the hardware execution on the component execution on the CPU: after processing each input packet received from \( f_{q_0} \), component \( C_1 \) delays its execution until it is provided with a new input from the hardware.
5.1 A linear chain where the first component is with deferred execution

The traceset of the file reader is:

\[ Tr(C_1) = \{ \exists f \text{false}, (\exists t, t \leq n, t \leq n) \} \]

where \( t \in Tr(C_1) \):

\[ Tr(C_1) = \{ \exists k > n, (\exists k \leq n, \exists c_1, \ldots, c_m, b_0) \}
\]

Given that we consider here no interference from the environment on the program variables, the trace that records the actual execution of the component is:

\[ (\exists true, bq_1, k := 1, \exists t, t \leq n) \]

We remind that one iteration of the trace above is denoted with \( t_{C_1} \). The execution of component \( C_1 \) is illustrated in Figure 5.3. Given the observed behaviour of the file reader and file writer components in practice where the sum of the deferral times is much smaller than the sum of the computation times of the processing actions of any other component in the chain, we adopt the condition
expressed below:

\[
2 \sum_{k=1}^{n} \tau_k < \sum_{j=1}^{m_0} \delta(c_j^i), \forall i, 1 < i \leq N
\]

(5.1)

5.2 Characterization of the unique trace \( \rho \)

The process of selecting the system trace is identical with the one presented in Chapter 2. At the end of this process we find that there exists a unique trace \( \rho \) that specifies the execution of the chain.

Lemma 5.1. Let \( C_i \) be such that \( (\forall j : j < i : P(C_j) > P(C_i)) \). Then there does exist a state \( t \) of \( \rho \) such that in any state \( s \) \( (t \subseteq s \bowtie bq_{i-1}! \subseteq \rho) \), \( C_j b bq_j? \) \([\text{in } s \text{ of } T_{cc}]\) for all \( j < i \).

Proof. We denote with \( Q(s) \) the following statement:

\[
Q(s) \overset{\text{def}}{=} (C_j b bq_j?[\text{in } s \text{ of } T_{cc}] \text{ for all } j < i)
\]

Let \( C_i \) be such that \( \forall j : j < i : P(C_j) > P(C_i) \). We define \( s_n, n \geq 1: \)

\[
s_n \sim bq_{i-1}!^n \subseteq \rho.
\]

A. As a first step we prove that if \( Q(s_n) \) holds then \( Q(s_{n+1}) \) holds as well. We prove this by analysis and construction of the trace \( u \), where:

\[
s_n \sim bq_{i-1}!^n \sim u \sim bq_{i-1}!^{n+1} \subseteq \rho
\]

After \( s_n \sim bq_{i-1}!^n \) is executed, the following actions follow in the trace:

- components \( C_j, 1 < j < i \) are de-blocked in cascade and execute one iteration of their loop after which they become blocked again at action \( bq_j? \). Only one iteration is possible because only one empty packet is available to process. We denote this trace with \( t_{d_a} \). \( C_i \) does not execute during \( t_{d_a} \) meaning that \( C_j, 1 < j < i \) become blocked at action \( bq_j? \) before \( C_i \) executes \( bq_{i-1}!^{n+1} \).
- \( C_1 \) is de-blocked as well leaving the deferral times to be used by:
  - communication actions of components \( C_j, 1 < j < i \) which have
been preempted during trace $t_i$: $f_{q_j}! - f_{q_{j-1}}?$. The preemption occurs when a component $C_j$ executes $b_{q_j}!$ and $P(C_{j-1}) > P(C_j)$. Note that these preempted components also have a lower priority than $C_1$, otherwise they would complete their loop iteration and become blocked again before $C_1$ starts executing.

- communication actions of $C_i$: $f_{q_i}!$. Note that by executing $f_{q_i}!$, $C_i$ de-blocks $C_{i+1}$ and potentially de-blocks in cascade all components $C_j$, $j > i$. We denote the trace recording the execution of these components with $t_R$. This de-blocking in cascade of components $C_j$, $j > i$, with $P(C_k) < P(C_1)$. $C_k$ executes interleaved with $C_1$ and because (5.1), $C_k$ consumes completely the deferral times of $C_1$. When $C_1$ resumes its execution, it progresses up until action $b_{q_j}!$ before $C_i$ started its $n+1$ iteration, therefore before $C_i$ executes $b_{q_i}!$.

- if $P(C_j) < P(C_i), \forall j > i$, the rest of the deferral times of $C_1$ is consumed by the execution of iteration $n+1$ of $C_i$. Given (5.1), follows that $C_i$ is not able to finish its processing action and execute $b_{q_{j-1}}!n+1$ before the end of the deferral times. This means that at the end of the last deferral time $C_i$ preempts $C_i$, finishes its loop iteration and becomes blocked again at action $b_{q_k}?$ before $C_i$ executes $b_{q_i}!n+1$.

The description above shows that $C_j$, $j < i$ become blocked again at action $b_{q_j}$ before $C_i$ executes $b_{q_i}!n+1$, meaning that $Q(s_{n+1})$ holds as well.

**B.** We wish to show by contraposition that for $n$ large enough, $Q(s_i)$ holds. Let us assume that $\neg Q(s_n), \forall n$. Given the definition of $s_n$ we have that

$$\#(s_{n+1}, f_{q_j}!) = \#(s_n, f_{q_j}!)+1, \forall n$$

If we can prove that

$$\neg Q(s_n) \land \neg Q(s_{n+1}) \Rightarrow \#(s_{n+1}, f_{q_0}?) > \#(s_n, f_{q_0}?) + 1, \forall n, \quad (5.2)$$

then the above two relations would imply that

$$\#(s_n, f_{q_0}?) - \#(s_n, f_{q_i}!) \text{ is strictly increasing as function of } n.$$ 

That would imply due to finite buffering that for sufficiently large $n$, $Q(s_i)$ holds which is a contradiction with the initial assumption.

Let us assume that $\neg Q(s_n), \forall n$. Then there exists $j < i$, $P(C_j) > P(C_i)$ where $C_j$ is not blocked at action $b_{q_j}$ in $s_n$ of $T_{cc}$. We set out to prove that in this
case (5.2) holds.

\textbf{a. - C}_j \textit{ is a data-driven component.}

1. We analyze first the case where \( C_j \) is a data-driven component preceding \( C_i \). If \( C_j \) is a data-driven component, then \( C_j \) cannot be ready-to-run in \( s_n \) of \( T_{ec} \). If that were so, then \( s_n \) could not be followed by action \( bq_{i-1} \) of \( C_i \) because \( P(C_j) > P(C_i) \). Hence \( C_j \) must be blocked. Given Property 2.2 and Property 2.3 follows that \( C_j \) must be blocked at action \( fq_{j-1} \).

We consider \( j \) minimum for which \( C_j \) is a data-driven component and \( C_j \) is not blocked at action \( bq_i \) in \( s_n \) of \( T_{ec} \). Follows \( C_j \) is blocked at action \( fq_{j-1} \).

Given Property 2.5 follows that all components \( C_k, 1 < k < j \) are blocked at action \( fq_{k-1} \).

2. The next question is where is \( C_1 \) blocked. We have so far that \( C_2 \) is blocked at action \( fq_1 \). Given Property 2.4 follows that \( C_1 \) cannot be blocked at \( bq_0 \). Since \( C_1 \) also cannot be blocked at \( fq_0 \) due to the cooperative environment, follows that when \( C_i \) executes, \( C_1 \) must be blocked from time perspective, during the deferral times. This means that when \( C_i \) executes, it executes only during the deferral times of \( C_1 \).

Consider the sub-trace between \( s_n \sim bq_{i-1}^n \) and \( s_{n+1} \sim bq_{i-1}^{n+1} \). Before \( bq_{i-1}^{n+1} \) is executed, \( c_i^1, \ldots, c_i^{n_i} \) must be executed first according to the program of \( C_i \). However from (5.1) we have that \( c_i^1, \ldots, c_i^{n_i} \) are completed only in at least two times the sum of the deferral times of \( C_1 \). This means that at the end of \( c_i^{n_i}, C_1 \) has completed its loop iteration at least twice, therefore it surely executed \( fq_0 \) two times. This implies

\[
\#(s_{n+1},fq_0) > \#(s_n,fq_0) + 1, \forall n
\]

We also have by definition of \( s_0 \)

\[
\#(s_{n+1},fq_1) = \#(s_n,fq_1) + 1, \forall n
\]

Hence

\[
\#(s_n,fq_0) - \#(s_n,fq_1) \text{ is strictly increasing as function of } n,
\]

which implies due to finite buffering that for sufficiently large \( n \), \( Q(s_n) \) holds which is a contradiction with the initial assumption.

\textbf{b. - C}_j \textit{ is the component with deferred execution (C}_1\text{).}

If \( C_j \) is \( C_1 \) then the reasoning continues as shown above as of \textbf{a-2}. \( \square \)

The converse statement of Lemma 5.1 is already given in Lemma 3.2: Let \( C_i \) be such that \((\forall j : j > i : P(C_j) > P(C_i))\) and consider a state \( s \) of \( St(\rho) \) such that the next action after \( s \) in \( \rho \) is one of \( A(C_i) \). Then \( C_j f_{q_{j-1}} \) in \( s \) of
5.2 Characterization of the unique trace \( \rho \)

\( T_{cc} \) for all \( j > i \).

We remind here that \( C_m \) denotes the component with minimum priority in the chain. The following theorem describes the execution of the overall system given based on the statements of the two previously mentioned lemmas (Lemma 5.1 and Lemma 3.2).

**Theorem 5.1.** Consider a pipeline system in which the first component is with execution deferral and the rest of components are data-driven. The system assumes a repetitive execution after a finite prefix \( t_{\text{init}} \). The entire execution is expressed in the following:

\[
\rho = t_{\text{init}} \preceq t_{\text{stable}}.
\]

and:

**A.** \( t_{\text{stable}} = (\prec t_1^{C_m} \prec t_{m-1}^{C_1} \prec t_{m-1}^{C_2} \prec t_{C_1}^{C_m} \prec t_{C_1}^{C_2} \prec t_{C_1}^{C_{m-1}C_{m-1}} \prec t_{C_1}^{C_m} \prec t_{C_1}^{C_{m-1}C_{m-1}} \prec t_{C_1}^{C_m} \prec t_{C_1}^{C_{m-1}C_{m-1}})_{00}, \)

if there exists \( C_i \in \text{RSC}_m, P(C_i) < P(C_1) \) (Figure 5.4),

**B.** \( t_{\text{stable}} = (\prec t_1^{C_1C_m} \prec t_{C_1}^{C_2} \prec t_{C_1}^{C_2} \prec t_{C_1}^{C_m} \prec t_{R}^{C_m})_{00} \)

otherwise (Figure 5.5).

Where

- \( t_1^{C_m} \) = \( t_{\text{true}} \preceq f_{m-1}^1 \preceq b_{qm}^m \preceq c_1^m \preceq \ldots \preceq c_m^{m} \preceq b_{m-1}^{m-1} \).
- \( t_{m-1}^{C_1} \) records the interleaved execution of components in \( \text{LSC}_m \) from \( C_{m-1} \) to \( C_2 \).
- \( t_{C_1}^{C_2} \) records an interleaved execution of \( C_1, C_j \) and \( C_m \). Component(s) \( C_j \) and \( C_m \) execute during the deferral times of \( C_1 \). Component(s) \( C_j \) have a lower priority than \( C_1 \). Traces \( t_2^{C_j} \) and \( t_2^{C_m} \) specify the actions of
$C_j$ and $C_m$ executed during the deferral times. Component(s) $C_j$ in $LSC_m$ have a lower priority than $P(C_1)$.

- $t_{C_j}^2 = \preceq fq_j! \quad \preceq f q_{j-1}! \preceq$.
- $t_{C_m}^2 = \preceq fq_m! \preceq$.
- $t_{C_m+1 \ldots C_{m-1}}$ records the interleaved execution of components in $RSC_m$ from $C_{m+1}$ to $C_{m-1}$. $C_i$ is the first component in $RSC_m$ with a lower priority than $C_1$, if such a component exists.
- $t_{C_1C_i}$ records an interleaved execution of component $C_1$ and any component $C_i$. Component $C_i$ executes during the deferral times of $C_1$.
- $t_{C_j}^1 = \preceq true \preceq f q_{j-1}! \preceq f q_j? \preceq c_1^j \ldots \preceq c_i^j \preceq, \forall i \geq m$.
- $t_{C_{j-1} \ldots C_N}$ records the interleaved execution of components in $RSC_m$ from $C_{j-1}$ to $C_N$.
- $t_{C_1C_m}$ records an interleaved execution of component $C_1$ and $C_m$. Component $C_m$ executes $t_{C_m}^1$ during the deferral times of $C_1$ and after $C_1$ is blocked.
- $t_R$ records the interleaved execution of components in $RSC_m$.

**Proof.** We use Lemma 5.1 and Lemma 3.2 for the component that holds the minimum priority overall the entire chain ($C_m$). According to these lemmas, there does exist a state $t \in St(\rho)$ in which all components in $LSC_m$ are blocked at action $b q_j$ [in $t$ of $\rho$] for all $j < m$, and all components $RSC_m$ are blocked at action $f q_{j-1}$ [in $t$ of $\rho$] for all $j > m$.

We prove by construction of sub-trace $t_{stable}$ that:

**A.** $t_{stable} = (\preceq t_{C_m}^1 \preceq t_{C_{m-1} \ldots C_2} \preceq t_{C_1C_m} \preceq t_{C_{m+1} \ldots C_{m-1}} \preceq t_{C_1C_i} \preceq t_{C_{i+1} \ldots C_N} \preceq)^o$

if there exists $C_i \in RSC_m$, $P(C_i) < P(C_1)$.

**B.** $t_{stable} = (\preceq t_{C_1C_m} \preceq t_{C_{m-1} \ldots C_2} \preceq t_{C_1C_jC_m} \preceq t_R \preceq)^o$

otherwise.

**A.** Consider $t$ a state in which all components in $LSC_m$ are blocked at action $b q_j$ [in $t$ of $\rho$] for all $j < m$, and all components $RSC_m$ are blocked at action $f q_{j-1}$ [in $t$ of $\rho$] for all $j > m$, and $t$ ends just before a new iteration of component $C_m$ is executed. This state exist according to Lemma 5.1 and Lemma 3.2. The execution of $\rho$ in time is depicted in Figure 5.4.

The first actions of $C_m$ following $t$ are described by $t_{C_m}^1$. When $C_m$ executes $b q_{m-1}!$, it causes a de-blocking in cascade of components in $LSC_m$. The interleaved execution of components $C_{m-1} \ldots C_2$ is recorded by trace $t_{C_{m-1} \ldots C_2}$ and is determined by priorities. The execution of the system so far is:

$$\preceq t \preceq t_{C_m}^1 \preceq t_{C_{m-1} \ldots C_2} \preceq$$
During this interleaved execution some components are preempted when they execute $bq_{j-1}!$. When component $C_1$ is deblocked, it executes interleaved with components from $LSC_m$ that have been preempted and that have a lower priority than $C_1$ ($C_j$), and with $C_m$. All these components ($C_j$ and $C_m$) will use the deferral times of $C_1$ to complete their iterations. The trace recording this execution is $t_{C_1C_jC_m}$. At the end of $t_{C_1C_jC_m}$ all components in $LSC_m$ have executed one iteration of their loop and became blocked at $bq_j$, $j < m$. The execution of the system so far is:

$$tt \sim t^1_{C_m} \sim t_{C_{m-1}C_2} \sim t_{C_1C_jC_m}$$

Once $C_m$ executes $fq_m$, during $t_{C_1C_jC_m}$, it causes a de-blocking in cascade of components in $RSC_m$. Note that this happens although $C_1$ did not yet complete its iteration. All components that have a higher priority than $P(C_1)$ will execute interleaved in the order determined by their priorities and their execution is recorded by $t_{C_{m+1}...C_{l-1}}$, where $C_i$ is the first component in $RSC_m$ with a lower priority than $C_1$. The execution of the system so far is:

$$tt \sim t^1_{C_m} \sim t_{C_{m-1}C_2} \sim t_{C_1C_jC_m} \sim t_{C_{m+1}...C_{l-1}}$$

After $t_{C_{m+1}...C_{l-1}}$, $C_1$ resumes its execution because it is the ready to run component with the highest priority at this state. $C_1$ continues its execution interleaved with $C_j$ which uses the deferral times. Given (5.1) follows that $C_j$ uses the entire length of the deferral times of $C_1$. The interleaved execution of $C_1$ and $C_j$ is recorded by $t_{C_1C_j}$. The execution of the system so far is:

$$tt \sim t^1_{C_m} \sim t_{C_{m-1}C_2} \sim t_{C_1C_jC_m} \sim t_{C_{m+1}...C_{l-1}} \sim t_{C_1C_j}$$

At the end of $t_{C_1C_j}$, $C_1$ has completed its iteration and became blocked again at $bq_j$. Note that at this point all components in $LSC_m$ are blocked at the action $bq_j$? [in $t$ of $p$] for all $j < m$.

When $C_i$ executes $fq_i!$, it de-blocks in cascade the rest of components in $RSC_m$. The interleaved execution of these components is recorded by $t_{C_i+1...C_N}$ and is entirely determined by their priorities. The execution of the system so far is:

$$tt \sim t^1_{C_m} \sim t_{C_{m-1}C_2} \sim t_{C_1C_jC_m} \sim t_{C_{m+1}...C_{l-1}} \sim t_{C_1C_i} \sim t_{C_{i+1}...C_N}$$

At the end of $t_{C_i+1...C_N}$ all components in $RSC_m$ have completed one iteration of their traces and have become blocked again at the action $fq_{j-1}!$ (and all components in $LSC_m$ are blocked already at $bq_j$). The only component ready to run in the system is $C_m$ which brings us at the similar situation at the end of $t_{\text{init}}$, therefore the execution described above repeats. Hence the
execution of the system during the stable phase is:

\[ t_{\text{stable}} = (\chi t_{C_m} - t_{c_{m-1}} - c_2 - t_{C_1} - c_m - t_{c_{m-1}} - c_{i-1} - t_{C_1} - c_i - t_{c_{i+1}} - c_m)^{\omega}. \]

**B.** Consider \( t \) a state in which all components in \( LSC_m \) are blocked at action \( bq_j \) [in \( t \) of \( \rho \)] for all \( j < m \), all components \( RSC_m \) are blocked at action \( fq_j \) [in \( t \) of \( \rho \)] for all \( j > m \), and \( t \) ends with action \( bq_m \) of \( C_m \). This state exist according to Lemma 5.1 and Lemma 3.2. The execution of the system proceeds as presented above until the end of trace \( t_{C_1} - c_{C_m} \). At the end of \( t_{C_1} - c_{C_m} \), all components in \( LSC_m \) have executed one iteration of their loop and are blocked again at \( bq_j \), \( j < m \). The execution of the system so far is:

\[ < t \sim t_{c_{m-1}} - c_2 - t_{C_1} - c_m \]

When \( C_m \) executes \( fq_m \) during the deferral time of \( C_1 \), it causes a de-blocking in cascade of components in \( RSC_m \). The interleaved execution of these components is recorded by trace \( t_R \). As opposed to the previous case, in case b, all components in \( RSC_m \) preempt \( C_1 \) because all components in \( RSC_m \) have a higher priority than \( P(C_1) \). Hence in this case the execution in cascade of components in \( RSC_m \) is uninterrupted by trace \( t_{C_1} - c_i \), \( P(C_i) < P(C_1), i > m \). All components in \( RSC_m \) execute one iteration of their loop and become blocked again at action \( fq_{i-1} \), \( m < i \leq N \). The execution of the system until this point is:

\[ < t \sim t_{c_{m-1}} - c_2 - t_{C_1} - c_m \sim t_R \]

At the end of \( t_R \), \( C_1 \) resumes its execution interleaved with a few actions of \( C_m \), namely with \( t_{C_1} \), according to the program of \( C_m \). This interleaved execution is recorded by \( t_{C_1} \). The execution of the system so far is:

\[ < t \sim t_{c_{m-1}} - c_2 - t_{C_1} - c_m \sim t_R \sim t_{C_1} \]
5.2 Characterization of the unique trace $\rho$

During $t_{C_1 C_m}$ executes $bq_{m-1}$ which causes the execution of the system to continue with $t_{m-1 \ldots C_2}^C t_{C_1 C_m}^C t_R^C$ as previously presented. The execution of the system so far is:

$$\tau^\omega = t_{C_1} \ldots t_{C_2} \sim t_{C_1 C_m} \sim t_R \sim t_{C_1} \ldots t_{m-1} \sim t_{C_1 C_m} \sim t_R^C$$

At the end of $t_R$, the situation above repeats, therefore we can specify $t_{\text{stable}}$ as:

$$t_{\text{stable}} = (\tau^\omega)^0$$

To show that our calculation of the system trace $\rho$ corresponds indeed to the execution of such a system in practice, we introduce below Figure 5.6 which illustrates the actual system execution over time, on one TriMedia VLWI processor. The system is a linear chain composed of eight components. In the figure T:FRID denotes the file reader component $C_1$, T:CMP1 denotes $C_2$, up to T:CMP7 denoting $C_8$. QFC1F denotes $fq_1$, QFC1E - $bq_1$, QC12F denotes $fq_2$, QC12E - $bq_2$, down to QC67F denoting $fq_7$, and QC67E - $bq_7$. Component $C_4$ has the lowest priority in the chain and $C_6$ is the first component in $RSC_4$ with a lower priority than $C_1$. The pattern of execution is shown between dashed lines. Note that at the beginning of each iteration of the stable phase all backward queues in $LSC_4$ are empty and all forward queues are empty in $RSC_4$. Also, the iteration starts with the execution of $C_4$ which de-blocks in cascade the components in $LSC_4$. During the first deferral time of $C_1$, $C_4$ de-blocks the components in $RSC_4$. $C_1$ is able to resume its execution only when the first component with a lower priority than $P(C_1)$ becomes ready to run - in this example $C_6$. Observe the interleaved execution of $C_1$ with $C_6$, and the de-blocking of the rest of $RSC_4$ after which the stable phase iteration ends.

5.2.1 Practical Applications

A first practical application of Theorem 5.1 is that, if given the computation times of components and the deferral times of $C_1$, the execution of the system (trace $\rho$ and associated schedule) is predicted at design time. From here, immediately follow values for the response time of the chain, the CPU utilization and the number of context switches occurring during the execution.

We present below a number of properties and corollaries that show how to achieve the optimization for CPU utilization, memory, chain response time and NCS. The following properties describe how to optimize CPU utilization by eliminating the idle times due to the execution with deferral of component $C_1$.

Property 5.1. When component $C_1$ is assigned the minimum priority in the chain, the deferral times of $C_1$ are idle.
Figure 5.6. **Stable phase execution in practice of a system composed of a component with deferred execution followed by 7 data-driven components.** CMP4 denotes the component with the lowest priority. CMP6 denotes the first component in RSC with a lower priority than the component with deferred execution.

**Proof.** When $C_1$ is assigned the minimum priority in the chain, it can execute only when all other components with a higher priority are blocked. This means that no component with higher priority will execute during the deferral times. Therefore the deferral times will not be used by the execution of any other component, hence the system experiences idle time during the deferral times. \[\square\]

**Property 5.2.** When $C_1$ does not have the minimum priority in the chain, idle times due to the execution with deferral of $C_1$ are eliminated.

**Proof.** When $C_1$ does not have the minimum priority in the chain, components with a lower priority can execute during the deferral times. Theorem 5.1 shows that the deferral times are used by $C_m, C_j$ ($\forall j < m, P(C_j) < P(C_1))$ and $C_i$. Given the condition expressed in (5.1), follows that the deferral times are completely used by the execution of these components. \[\square\]

Figure 5.7 shows that idle times occur when $C_1$ is assigned minimum priority. In the figure, T:FRID denotes the file reader (component $C_1$), and T:IDLE represents the IDLE task. Note that the idle times indicated by the red blocks in the figure, fit in the deferral times of T:FRID.

Corollaries presented in Chapter 3 addressing the optimization of memory,
and NCS for a chain composed of only simple data-driven components hold here as well. The proofs are identical.

- the minimum necessary and sufficient capacity of each queue in the chain to avoid deadlock is 1. (Corollary 3.3).

- the number of context switches during the stable phase is minimal when $P(C_1) < \ldots < P(C_N)$ (Theorem 3.2).

Note that when the deferral times of $C_1$ are used by a component with a lower priority than $P(C_1)$ as suggested by Property 5.2, NCS is not optimal because $2k$ more context switches occur.

**Tradeoff** - The priority assignment suggested by Property 5.2 implies that optimal utilization of CPU is achieved by eliminating the idle times during the deferred execution of $C_1$, but at the cost of additional $2k$ context switches.

In practice trading the overhead introduced by the $2k$ context switches for an optimal CPU utilization against is profitable because the overhead introduced by context switches is much less than the duration of the deferral times.

The corollary below shows how to achieve best chain response time in the conditions that no idle time occurs during the execution of the system:

**Corollary 5.1.** The optimal chain response time under the condition of no idle time during the execution of the chain is achieved when $C_2$ is assigned the minimum priority and $P(C_2) < P(C_3) < \ldots < P(C_{N-1}) < P(C_N)$.

**Proof.** Theorem 3.3 implies that for sub-chain $C_2 \ldots C_N$ composed of only data-driven components the minimum response time is to be achieved when $P(C_2) = \min_{i=2}^{N} P(C_i)$ and $P(C_2) < P(C_3) < \ldots < P(C_{N-1}) < P(C_N)$. Property 5.2 shows that in order or eliminate idle times during the execution of the chain $C_1$ cannot have the minimum priority, therefore $P(C_1) > P(C_2)$. $\square$
5.3 Adding a time-driven component at the end of the chain

In this section we study the influence of a component with deferred execution on the execution of a chain with timing constraints as described in section 4.1. The chain we analyze consists of a component with deferred execution followed by \( N - 2 \) simple data-driven components, and ends with a time-driven component as in Figure 5.8. As in the previous case described above, the process of selecting the trace that specifies the execution of the system, as well as the proof with respect to the unicity of this trace \((\rho)\) are identical with the one presented in section 2.6.3.

We reiterate the condition used in section 4.1.2, \( S^M < T_N \). The following lemma states that for the type of chain we study in this section, if \( S^M < T_N \), after a finite prefix \( t_{\text{init}} \), all components \( C_i, 1 \leq i < N \) will be blocked at action \( b_q^i \). We mention here that \( S^M \) includes in this case also the duration of the deferral times of component \( C_1 \). That is because when \( C_1 \) has the minimum priority in the chain the deferral times are not used by any other component, hence they must be accounted for. That is because they delay the execution end of each iteration of \( C_1 \), hence the production of a packet in \( f_q^2 \).

\[
S^M = \sum_{i=1}^{N} S^M_{b_i}(t_{c_i}) + \sum_{k=1}^{n} \tau_k.
\]

**Lemma 5.2.** Given \( S^M < T_N \), there exists \( t_{\text{init}} \in St(\rho) \) such that \( C_i \ b_q^i \) [in \( t_{\text{init}} \) of \( T_{cc} \)] for all \( i \), \( 1 \leq i < N \).

**Proof.** Lemma 5.1 implies that there does exist a state \( t \) of \( \rho \) such that in any state \( s, t \subseteq s^{-} b_{q_m-1} \subseteq \rho, C_i \ b_q^i \) [in \( s \) of \( T_{cc} \)] for all \( i < m \). This means that in state \( t \) all backward queues \( b_{q_i} \), \( \forall 1 \leq i < m \) are drained \((L(b_{q_i}) = 0, \forall 1 \leq i < m)\).

Theorem 5.1 shows that after \( t \), \( LSC_N \) executes according to a repetitive pattern of execution during which each component executes one iteration of their loop after their become blocked again at action \( b_q^i \).

![Figure 5.8. Chain composed of a component with deferred execution, \( N - 2 \) data-driven components and a time-driven component.](image-url)
5.3 Adding a time-driven component at the end of the chain

$S^M < T_N$ implies that the rate of production of packets in $fq_{N-1}$ is higher than the rate of consumption. This means that after a finite number of $T_N$, $fq_{N-1}$ will be filled to its capacity ($bq_{N-1}$ drained), and $C_{N-1}$ becomes blocked at action $bq_{N-1}?$.

From here on $C_{N-1}$ is dependent on $C_N$ to release an empty packet which happens once every $T_N$. Hence from here on $C_{N-1}$ executes one iteration of its loop once every $T_N$, and therefore also consumes during each $T_N$ only one full packet from $fq_{N-1}$.

Within each $T_N$, $LSC_{N-1}$ continues to execute according to a repetitive pattern as shown in Theorem 5.1. Again $S^M < T_N$ implies that after a finite number of $T_N$, $fq_{N-2}$ will be filled to its capacity ($bq_{N-2}$ drained), and $C_{N-2}$ becomes blocked at action $bq_{N-2}?$.

The process repeats until a state (denoted with $t_{\text{init}}$) where all backward queues $bq_i, \forall m \leq i \leq N$ are drained ($L(bq_i) = 0, \forall m \leq i \leq N$).

The following theorem specifies the execution of the system studied here. The main difference in system behaviour when comparing with the system behaviour described in Theorem 5.1 is that in the present case $C_i$ is limiting the outputing of packets to the environment due to its time-driven execution.

**Theorem 5.2.** Consider a system as in Figure 5.8. When $S^M < T_N$, the system assumes a repetitive, periodical behavior after a finite initialization phase. The complete behavior illustrated in Figure 5.9 is characterized by

\[ \rho = t_{\text{init}} \sim t_{\text{stable}}. \]

\[ t_{\text{stable}} = (t_{C_N} \sim t_L \sim d(T_0 + i * T_N))^{\omega}. \]

Where:

- $t_{C_N} = \check{\gamma} fq_{N-1} \sim bq_N \sim c_1^{N} \sim ... \sim c_{N-1}^{N} \sim bq_{N-1} \sim fq_N \sim i := i + 1$. \footnote{This is a recursive definition.}
- $t_{L}$ records the interleaved execution of components in $LSC_N$:
  \[ t_L = t_{C_N-1 \ldots C_2} \sim t_{C_1 C_j}. \]
- $t_{C_{N-1 \ldots C_2}}$ records the interleaved execution of components from $C_{N-1}$ to $C_2$.
- $t_{C_1 C_j}$ records an interleaved execution of $C_1$ and $C_j$, where $P(C_j) < P(C_1)$. The number of components $C_j$ with a lower priority than $C_1$ that execute during the deferral times of $C_1$ depends on the relation between the computation times of the communication actions of these components and the deferral times of $C_1$.
  \[ t_{C_j}^2 = \check{\gamma} fq_j \sim fq_{j-1} \sim ... \sim fq_1 \sim \check{\gamma}. \]
- $SQoS(t_{\text{stable}}, t_{\text{init}}, fq_{N-1}?, fq_N!, T_0^N, T_N)$ holds.
Proof. We prove the repetitive nature of system by construction of the sub-trace \( t_{stable} \).

Lemma 5.2 shows that at the end of \( t_{init} \) all components in \( LSC_N \) are blocked at action \( bq_i \), \( 1 \leq i < N \).

After \( t_{init} \) the only component that can execute is \( C_N \). \( C_N \) executes \( t_{C_N} \) according to its program. \( t_{C_N} \) represents an entire iteration of its loop except the delay action. Sub-trace \( t_{C_N} \) contains action \( bq_{N-1} \) which de-blocks \( C_{N-1} \).

Given that after \( bq_{N-1} \) the \( bq_{N-1} \) action of \( C_{N-1} \) becomes ready, implies that the delay action \( d(T_0 + l * T_N) \) is postponed. At this point trace \( \rho \) can be expressed as:

\[
\rho = t_{init} \circ t_{C_N} \circ u.
\]

The de-blocking of \( C_{N-1} \) is followed by a de-blocking in cascade of components in \( LSC_N \), down to and including \( C_2 \). The order of execution of these components is completely determined by their priorities and is recorded by \( t_{C_{N-1} \ldots C_2} \). Depending on the priority assignment, some of these components have been preempted during \( t_{C_{N-1} \ldots C_2} \) after action \( bq_j \), \( 1 < j < N \), others have completed an iteration of their loop and became blocked again at action \( bq_j \). At this point trace \( \rho \) can be expressed as:

\[
\rho = t_{init} \circ t_{C_N} \circ t_{C_{N-1} \ldots C_2} \circ u.
\]

When \( C_1 \) is de-blocked, only those component \( C_j \), \( 1 < j < N \) with a lower priority than its own \( (P(C_j) < P(C_1), 1 < j < N) \) are still ready. Given that \( C_1 \) has a higher priority, these components will execute during its deferral times. Each component \( C_j \) finishes one iteration of its loop by executing \( t_{C_j}^2 = \{fq_j \} \circ fq_{j-1} \circ \}, \) after which they become blocked at action \( bq_j \). \( C_1 \) as well completes one iteration of its loop and becomes blocked at \( bq_j \). The interleaved execution of \( C_1 \) and components \( C_j \) where \( P(C_j) < P(C_1) \), \( 1 < j < N \) is recorded by trace \( t_{C_1 \ldots C_j} \). So far trace \( \rho \) can be expressed as:

\[
\rho = t_{init} \circ t_{C_N} \circ t_{C_{N-1} \ldots C_2} \circ t_{C_1 \ldots C_j} \circ u.
\]

or as

\[
\rho = t_{init} \circ t_{C_N} \circ t_{C_{N-1} \ldots C_2} \circ t_{C_1 \circ C_j} \circ u.
\]

Given that \( S^M < T_N \), the execution of components in \( LSC_N \) recorded by \( t_L \) ends before \( T_0 + l * T_N \). Also given that at this point no other actions are ready, the delay action \( d(T_0 + l * T_N) \) follows in the trace:

\[
\rho = t_{init} \circ t_{C_N} \circ t_L \circ d(T_0 + l * T_N) \circ u.
\]

\( \sigma(t_{init} \circ t_{C_N} \circ t_L) < T_0^N + l * T_N \) also implies that the effect of the delay action is to advance time until moment \( T_0^N + l * T_N \). This implies that

\[
\sigma(t_{init} \circ t_{C_N} \circ t_L \circ d(T_0^N + l * T_N)) = T_0^N + l * T_N
\]  \( (5.3) \)
At time $T_0 + i \ast T_N$, $C_N$ is ready to run, and all components in $LSC_N$ are blocked at action $bq_i$, $1 \leq i < N$. This situation is similar with the one at the end of $t_{init}$ which implies that the execution of the system will repeat as long as there is input. Assuming infinite input, the sub-trace following $t_{init}$ is infinitely repetitive:

$$\rho = t_{init} \prec (\exists t_{C_N} \prec t_L \prec d(T_0 + i \ast T_N)X)^\omega.$$ 

Therefore $\rho$ can be expressed as

$$\rho = t_{init} \prec t_{stable}.$$  

where

$$t_{stable} = (\exists t_{C_N} \prec t_L \prec d(T_0 + i \ast T_N)X)^\omega.$$ 

The repetitive execution expressed above and (5.3) show that for each iteration of $C_N$ during $t_{stable}$:

$$\sigma(s_1 \prec f_{q_{N-1}}^{\neg k}) - \delta(f_{q_{N-1}}^{\neg k}) \geq T_0^N + k \ast T_N,$$

$$\forall s_1 \prec f_{q_{N-1}}^{\neg k} \in St(\rho), t_{init} \subseteq s_1, \forall k \in \mathbb{N}$$

Also given (4.1) we have that

$$\sigma(s_2 \prec f_{q_N}^{\neg k}) < T_0^N + k \ast T_N + flush, \forall s_2 \prec f_{q_N}^{\neg k} \in St(\rho), t_{init} \subseteq s_2, \forall k \in \mathbb{N}$$

This implies that $SQoS(t_{stable}, t_{init}, f_{q_{N-1}}^{\neg}, f_{q_N}^{\neg}, T_N)$ holds.

Figure 5.9. Execution in time of system composed of a component with deferred execution, $N - 2$ data-driven components and ending with a time-driven component.

The Lemma below shows that the QoS requirement is also satisfied during the initial phase $t_{init}$.

**Lemma 5.3.** $S \prec T_N \wedge P(C_N) = \max_{i=1}^N P(C_i) \Rightarrow$

$$SQoS(t_{init}, \varepsilon, f_{q_{N-1}}^{\neg}, f_{q_N}^{\neg}, T_0^N, T_N).$$

**Proof.** Identical with the one presented in Lemma 4.2. \[\square\]
The corollary below states that the QoS requirement as defined in section 4.1.1 is satisfied for the entire execution of the chain (entire trace $\rho$) if at design time it is ensured that $SM < TN$ and $CN$ is assigned the highest priority in the chain.

**Corollary 5.2.** $S < TN \quad \land \quad P(C_N) = \max_{i=1}^{N} P(C_i) \quad \Rightarrow \quad S_{Qos}(\rho, \varepsilon, fq_{N-1}, fq_N, T^1_0, T_N)$.

**Proof.** Results directly from Lemma 5.3 and Theorem 5.2. \hfill $\square$

Lemma 5.2 can be easily extended to cover the case where the time-driven component executes according to the interlaced standard. We present below the corresponding lemma and Stable Phase Theorem for this case.

**Lemma 5.4.** Given $SM < 2TN$, there exists $t_{init} \in St(\rho)$ such that $C_i \ b qi \ [in t_{init} of \ T_c \ ]$, for all $i, 1 \leq i < N$. \hfill $\square$

We denote with $MSC$ the sub-chain obtained by eliminating $C_1$ and $C_N$ from the initial chain. Also $SM_{MSC}$ is the sum of the worst case computation times of one loop iteration of components $C_2, \ldots, C_{N-1}$:

$$SM_{MSC} = \sum_{i=2}^{N-1} SM(t_C).$$

**Theorem 5.3.** When $SM < 2TN$, the pipeline system in which the first component is with execution deferral and the last component has a periodic behavior (interlaced standard), assumes a repetitive, periodical behavior after a finite initialization phase. The complete behavior is characterized by

$$\rho = t_{init} - (T^1_{CN} - t^1_L - d(T^N_0 + i \cdot TN) - t^1_CN - t^2_CN - d(T^N_0 + i \cdot TN) + \omega)$$

as illustrated in Figure 5.10 where

![Figure 5.10. Execution of a chain composed of a component with deferred execution, $N-2$ data-driven components, and ending with a time-driven component.](image-url)
5.3 Adding a time-driven component at the end of the chain

- $t^1_L$ and $t^2_L$ are sub-traces recording the concurrent execution of components in sub-chain LSCN.
- $t^{C_i,C_j}$ records an interleaved execution of $C_i$ and $C_j$, $P(C_j < P(C_i))$. Depending on the computation times of the communication actions of Components $C_j$ have a lower priority than $C_i$ and execute during the deferral times of $C_i$.
- Trace $t^{1,C_i,C_j}$ and $t^{2,C_i,C_j}$ are sub-traces of $t^{C_i,C_j}$.
- Sub-trace $t_{L_1}$ records the interleaved execution of components in LSCN between $t^{1}_{C_N}$ and $d(T^N_0 + i * T_1)$. Sub-trace $t_{L_2}$ records the interleaved execution of components in LSCN between $t^{2}_{C_N}$ and $d(T^N_0 + i * T_1)$ (of the next iteration $i$) when $T_N < S^M < 2T_N$. Trace $t_{L_2}$ is empty when $S^M < T_N$.
- When $S^M < T_N$, $t_{L_1} = u^< t^{C_i,C_j}$.
- When $T_N < S^M < 2T_N$ and $S^M_{MSC} > T_N$, $t_{L_2} = u^= t^{C_i,C_j}$.
- When $T_1 < S^M < 2T_1$ and $S^M_{MSC} < T_N$, $t_{L_1} = u^< t^{C_i,C_j}$ and $t_{L_2} = t^{2,C_i,C_j}$.
- During one iteration of $t_{stable}$ all components execute one iteration of their loop. At the end of each iteration of $t_{stable}$ components in LSCN are blocked at $bq_i^?$, $1 \leq i < N$.

**Proof.** The approach of the proof is similar to the one of Theorem 4.2. □

5.3.1 Practical Applications

Practical applications for this case inherit design criteria both from the case described in the previous section and from the case described in section 2.6.3 describing a chain of data-driven components and one time-driven component at the end of the chain.

Due to the timing properties of the chain execution satisfying condition $S^M < T_N$, we can conclude again that regardless of its priority, component $C_N$ has the same effect on the stable phase trace of this chain, as a data-driven component with minimum priority has in the case of a chain composed of only data-driven components and a component with deferred execution (described in section 5.2). Owing to this fact, we find that corollaries addressing the stable phase of a chain described there (with $C_N$ having the lowest priority) hold in this case as well. From here we deduce that:

- the minimum necessary and sufficient capacity of each queue in the chain is 1 (Corollary 3.3).
- as shown in Theorem 3.3 the response time of the chain is reduced by reducing the capacities of queues preceding $C_N$. 

Proof. The approach of the proof is similar to the one of Theorem 4.2. □

5.3.1 Practical Applications

Practical applications for this case inherit design criteria both from the case described in the previous section and from the case described in section 2.6.3 describing a chain of data-driven components and one time-driven component at the end of the chain.

Due to the timing properties of the chain execution satisfying condition $S^M < T_N$, we can conclude again that regardless of its priority, component $C_N$ has the same effect on the stable phase trace of this chain, as a data-driven component with minimum priority has in the case of a chain composed of only data-driven components and a component with deferred execution (described in section 5.2). Owing to this fact, we find that corollaries addressing the stable phase of a chain described there (with $C_N$ having the lowest priority) hold in this case as well. From here we deduce that:

- the minimum necessary and sufficient capacity of each queue in the chain is 1 (Corollary 3.3).
- as shown in Theorem 3.3 the response time of the chain is reduced by reducing the capacities of queues preceding $C_N$. 

Proof. The approach of the proof is similar to the one of Theorem 4.2. □
• the number of context switches during the stable phase can be reduced by assigning priorities as $P(C_1) < P(C_2) < ... < P(C_{N-1})$ and $Cap(f_{q_i}) = 2 \forall i, 1 \leq i < N - 1$ (Theorem 3.2).
• Elimination of idle times during the execution deferral is achieved by not assigning $C_1$ the minimum priority as shown in Property 5.2

5.4 A linear chain ending with a component with deferred execution

In this section we study the case of a linear chain consisting of a time-driven component, a number of data-driven components and ending with a component with deferred execution as shown in Figure 5.11. The program of component $C_N$ is presented in Figure 5.12. The traceset of $C_N$ is:

```
C_N : while(true) do
{  
    receive(f_{q_{N-1}}.p);
    k := 1;

    C_N^p : while(k \leq n) do
    {  
      receive(b_{q_N}.q); process.f vr(p,q); send(f_{q_N}.q);
      getTime(T^0_0); delay._wait(T^0_0 + t_1); k := k + 1;
    }
    send(b_{q_{N-1}}.p);
}
```

Figure 5.12. Program of a component with deferred execution at the end of the chain.

$Tr(C_N) = \{ \langle \text{false} \rangle, \langle \text{true} \rangle \rightarrow f_{q_{N-1}}? \rightarrow k := 1 \sim t \sim b_{q_{N-1}}! \langle \text{false} \rangle^0 \}$. 
where \( t \in Tr(C_N^k) \):
\[
Tr(C_N^k) = \{ \forall k > n^k, \\
\forall k \leq n \vdash bq_N ? \leftarrow c_1^k \leftarrow \ldots \leftarrow c_{m_N}^k \leftarrow f q_N ! \leftarrow \\
gt(T^N_0) \leftarrow d(T^N_0 + \tau_k) \leftarrow k := k + 1 \leftarrow k > n \not\vdash, \\
\vdots \\
(\forall k \leq n \vdash bq_N ? \leftarrow c_1^k \leftarrow \ldots \leftarrow c_{m_N}^k \leftarrow f q_N ! \leftarrow \\
gt(T^N_0) \leftarrow d(T^N_0 + \tau_k) \leftarrow k := k + 1 \not\vdash) \} \}
\]
Given that the environment does not interfere on the program variables, the trace that records the actual execution of the component is:
\[
(\forall true \vdash f q_{N-1} ! ? k := 1 \leftarrow \\
(k \leq n \vdash bq_N ? \leftarrow c_1^k \leftarrow \ldots \leftarrow c_{m_N}^k \leftarrow f q_N ! \leftarrow \\
gt(T^N_0) \leftarrow d(T^N_0 + \tau_k) \leftarrow k := k + 1 \not\vdash) \} \}
\]
An example from practice of such a component is the file writer component which writes on a disk the data that receives as input. An example from practice of the type of chain we study here would be a video surveillance application where the images captured by a camera are improved by the processing of the data-driven components and finally are stored on the disk by the file writer.

The QoS requirement of this system coincides with the QoS requirement of the system analyzed in section 4.3 in the case of a chain starting with a time-driven component holds here as well. We show in the theorem and corollary below that the mechanism \((SM < 2T_1)\) for satisfying this requirement and the repetitive behaviour of the system when this mechanism is implemented hold again. \(SM\) takes into account also the deferral times of \(CN\).

**Theorem 5.4.** Consider a system as in Figure 5.11. When \(SM < 2T_1\) the pipeline system assumes a repetitive, periodical behavior after a finite initial phase. The complete behavior is characterized by
\[
\rho = t_{init} \leftarrow (\forall t_{C_i} \leftarrow t_{R_i} \leftarrow d(T^1_0 + i * T_1) \leftarrow t^{1^i}_{C_i} \leftarrow t_{R_2} \leftarrow d(T^1_0 + i * T_1)) \not\vdash\}
\]
where
\[
t_{init} = \forall gt(T^1_0) \leftarrow i := 0 \leftarrow \\
fq_0 ? \leftarrow bq_1 ? \leftarrow c_1^{1^i} \leftarrow \ldots \leftarrow c_i^{m_i} \leftarrow bq_0 ! \leftarrow d(T^1_0 + i * T_1) \not\vdash,
\]
as illustrated in Figure 5.13 and:

- **Sub-traces** \(t_{R_i}\) and \(t_{R_2}\) record the interleaved execution of components in \(RSC_1\).
• Sub-trace $t_{R_1}$ records the interleaved execution of components in $RSC_1$ between $t_{C_1}^1$ and $d(T_0^1 + i \cdot T_1)$.

• Sub-trace $t_{R_2}$ records the interleaved execution of components in $RSC_1$ between $t_{C_1}^1$ and $d(T_0^1 + i \cdot T_1)$ (of the next iteration $i$) when $T_1 < S^M < 2T_1$. Trace $t_{R_2}$ is empty when $S^M < T_1$.

• Trace $t_{C_1/C_j}$ records the interleaved execution of $C_N$ with other components $C_j$ where $P(C_j) < P(C_N)$. The deferral times of $C_N$ are used by components $C_j$, $1 < j < N$.

• Trace $t_{C_1/C_j}^1$ and $t_{C_1/C_j}^2$ are sub-traces of $t_{C_1/C_j}$.

• When $S^M < T_1$, $t_{R_1} = u \circ t_{C_1/C_j}$.

• When $T_1 < S^M < 2T_1$ and $S^M_{MSC} \geq T_1$, $t_{R_2} = u \circ t_{C_1/C_j}$.

• When $T_1 < S^M < 2T_1$ and $S^M_{MSC} < T_1$, $t_{R_1} = u \circ t_{C_1/C_j}^1$ and $t_{R_2} = t_{C_1/C_j}^2$.

• Trace $t_{C_1/C_j}$ records the interleaved execution of $C_N$ with other components $C_j$ where $P(C_j) < P(C_N)$. The deferral times of $C_N$ are used by components $C_j$, $1 < j < N$.

• Trace $t_{C_1/C_j}^1$ and $t_{C_1/C_j}^2$ are sub-traces of $t_{C_1/C_j}$.

• During one iteration of $t_{stable}$ all components execute one iteration of their loop. At the end of each iteration of $t_{stable}$ components in $RSC_1$ are blocked at $f q_i-1 ?$, $1 < i \leq N$.

Proof. The approach of the proof is similar to the one of Theorem 4.3, the statements of the theorem are proved by construction of the trace $\rho$. For this reason we present here only a sketch of the proof. During the initial phase only $C_1$ executes because it is the only ready component - the other components are blocked at action $f q_i-1 ?$, $i > 1$. Hence given the trace iteration of $C_1$.

![Figure 5.13](image-url) Execution of a chain composed of a time-driven component, $N-2$ data-driven components, ending with a component with deferred execution.
5.4 A linear chain ending with a component with deferred execution

\[ t_{\text{init}} = \frac{\xi}{t} \cdot g(T_0^1) \sim i := 0 \sim f_{q_0} \sim b_{q_1} \sim c_1' \sim \ldots \sim c_{n_{\text{nt}}} \sim b_{q_0}! \sim d(T_0^1 + i \cdot T_1) \frac{\xi}{t}. \]

When \( C_1 \) during \( t_{C_1}^2 \) executes \( f_{q_1}! \) at the end of \( t_{\text{init}} \) it causes a de-blocking in cascade of components in \( \text{RSC}_1 \). Each of these components execute one iteration of their loop after which they become blocked at action \( f_{q_i - 1} \), \( i > 1 \) again. The interleaved execution of these components is recorded by traces \( \xi_t \) and \( t_{R_2} \). The order of actions in each of the traces \( \xi_t \) and \( t_{R_2} \) is determined by the priority assignment. During the interleaved execution some components will be preempted when they execute action \( f_{q_i} \), \( 1 < i < N \). Components that have been preempted and have a lower priority than \( C_N \) will execute interleaved with \( C_N \) and make use of the deferral times.

The interleaved execution of components in \( \text{RSC}_1 \) is interrupted every \( T_0^1 + i \cdot T_1, i = 2p - 1 \) \( p > 0 \) by the odd period number execution of \( C_1 \) recorded by \( t_{C_1}^1 \). This is due to the fact that \( C_1 \) has the highest priority in the chain. Given \( S_M^M < 2 \cdot T_1 \) follows that the interleaved execution of components in \( \text{RSC}_1 \) always ends before moment \( T_0^1 + i \cdot T_1, i = 2p \) \( p > 2 \) when \( t_{C_1}^2 \) is due to execute. This means that before \( t_{C_1}^2 \) is executed again, each of the components in \( \text{RSC}_1 \) execute one iteration of their loop after which they become blocked at action \( f_{q_i - 1} \), \( i > 1 \) again. This is an identical situation as the one at the end of the initial phase, hence the execution of the system repeats.

A few cases need to be distinguished with respect to the interleaved execution of components in \( \text{RSC}_1 \):

**A.** When \( S_M^M < T_1 \), the interleaved execution of components in \( \text{RSC}_1 \) ends before moment \( T_0^1 + i \cdot T_1, i = 2p - 1 \) \( p > 0 \). Therefore the interleaved execution of components in \( \text{RSC}_1 \) always ends before the execution of \( t_{C_1}^1 \). This means that \( t_{R_1} \) ends with \( t_{C_N C_j} \):

\[ t_{R_1} = u \sim t_{C_N C_j}. \]

It also means that \( t_{R_2} \) is empty.

**B.** When \( T_1 < S_M^M < 2T_1 \) and \( S_{\text{MSC}}^M \geq T_1 \), the interleaved execution of components in \( \text{MSC} \) exceeds moment \( T_0^1 + i \cdot T_1, i = 2p - 1 \) \( p > 0 \) and \( t_{R_2} \) ends with \( t_{C_N C_j} \):

\[ t_{R_2} = u \sim t_{C_N C_j}. \]

**C.** When \( T_1 < S_M^M < 2T_1 \) and \( S_{\text{MSC}}^M < T_1 \), the interleaved execution of components in \( \text{MSC} \) ends before moment \( T_0^1 + i \cdot T_1, i = 2p - 1 \) \( p > 0 \) and \( t_{R_1} \) contains only part of \( t_{C_N C_j} \) which denotes the interleaved execution of \( C_N \) and components with lower priority than \( P(C_N) \). This partial execution is denoted with \( t_{C_N C_j}^1 \):

\[ t_{C_N C_j}^1 = \]
Obviously $t_{R_2}$ contains the rest of the interleaved execution between $C_N$ and components with a lower priority denoted with $\mathcal{E}_{C, j}$:

$$t_{R_2} = t^2_{C, j}$$

We close this chapter by mentioning the following corollary which states that the QoS requirement is satisfied given that $S^M < 2T_1$ and $C_1$ is assigned the highest priority in the chain. Notice here that compared to the system composed of a time-driven component ($C_1$) and $N - 1$ data-driven components (section 4.3), in the current case, the fact that $C_N$ is a component with deferred execution does not imply that more conditions need to be met at design time such that the QoS requirement is satisfied.

**Corollary 5.3.** $S^M < 2T_1 \land P(C_1) = \max_{i=1}^{N} P(C_i) \Rightarrow$

$$SQoSci(\rho, \varepsilon, f^{q_0}_1, b^{q_0}_1, f^{q_0}_0, b^{q_0}_0, f^{q_1}_1, f^{q_1}_0, T_0^1, T_1).$$

**Proof.** Follows directly from Theorem 5.4.

### 5.4.1 Practical Applications

The practical applications presented in subsection 4.3.3 hold here as well for the same reasons they held in the previous case.

We make the observation here that because $C_N$ being the last component in the chain will be de-blocked only when all the other components have been de-blocked and executed one iteration of their loop. By following a similar reasoning as shown in Property 5.1 and Property 5.2 follows again that CPU utilization by using the deferral times of $C_N$ is improved by not assigning $C_N$ the lowest priority in the chain.

### 5.5 Summary

We have presented in this chapter a study of systems that include components with deferred execution. Examples of such components from the TSSA architecture are the file reader and file writer.

We adopt an incremental approach starting from a system without timing constraints to systems with timing constraints that also include components with deferred execution.

We show that the traces that record the execution of the systems we analyzed adopt a repetitive pattern given that the sum of the deferral times is
always smaller than the sum of the worst case computation times for one iteration of any of the other components in the chain.

Also given that the sum of the worst computation times for one iteration of all components in the chain is smaller than the period of the time-driven component, the quality of service requirements of the systems with timing constraints are satisfied.

In terms of practical applications of the analysis in this chapter we show again how to optimize at design time the amount of resources (memory and CPU) used by the system, the response time of the chain, the number of context switches. The new addition in terms of design criteria for optimization compared to the other chapters regards the optimization of CPU utilization, such that amount of idle time during the execution of the system is at a minimum by making \( P(C_1) \) not minimal.
In this chapter we study the influence of the input stream contents on the overall execution of a media processing system. This influence comes as a result of the fact that the behaviour of some components in the system changes depending on the input stream contents. Typical examples of such components from practice include the video/audio encoder and decoder.

In section 6.1 we explain the behavior of such a component, we show the program and the traceset associated with the component program.

Building towards realistic systems, we adopt an incremental approach starting from a system without timing constraints to systems with timing constraints. The aim is to study the influence of the component with execution dependent on the input stream content on the overall systems behaviour.

To that end section 6.2 presents the analysis of the execution of a linear chain consisting of $N$ components: a component with deferred execution, a component with execution dependent on the contents of the input stream, and $N - 2$ data-driven components. Subsequently, in section 6.3 we introduce timing constraints to the system we analyzed in section 6.2. In both cases we show that the traces that record the execution of the systems we analyzed adopt a repetitive pattern dependent on the content of the input stream. The two pat-
terns correspond to the two execution scenarios of the component dependent on contents of the input stream.

The closest resembling systems studied in previous chapters are presented in sections 5.2 and 5.3. In terms of the composition of the systems, the difference with the systems presented in Chapter 5 is the addition of the component with dependency on the input stream contents. As a similarity in terms of the system execution, as in Chapter 5, the execution of the system with timing constraints is driven by the time-driven component as well. The differences with respect to the system execution appear when analyzing the patterns of execution - a simple pattern as in all previous chapters as opposed to a parameterized pattern in this chapter.

Finally, in section 6.4 we use the case study of a video decoding chain to present and support our analysis of these type of systems with experimental data. The video decoding chain we analyze consists of a file reader component, a video decoder component, a number of simple data-driven components, and ends with a video renderer component. The role of the simple data-driven components following the video decoder in the chain is to improve the quality of the decoded frame produced by the video decoder. An example of such a component is the sharpness enhancement component. The video renderer is used to display each decoded frame at a fixed rate, and therefore the video renderer is a periodic task. The program and traceset of the video render executing according to the interlaced standard is shown in section 4.2. The analysis regarding the file reader component is described in Chapter 5.

Finally section 6.5 addresses resource optimization issues and shows how practical applications presented in previous chapters are valid in this case as well.

6.1 A component with execution dependent on the input stream contents

To give an insight into what determines the dependency of the video decoder execution on the input stream contents, we explain the behaviour of this component. In the case of the video decoder, the input-output relation is determined by how many encoded frames are contained in each new input FP received by the component. If the new FP contains \( x \) encoded frames, then for this input FP, the video decoder will produce \( x \) FPs (decoded frames). The input-output relation of the component in this case is \( 1 : x \). If on the contrary \( x \) new input FPs contain only one encoded frame, then the input-output relation is \( x : 1 \) because the video decoder needs to receive \( x \) input FPs in order to produce 1 FP (decoded frame). Therefore the nature of the input video stream (the sizes of
6.1 A component with execution dependent on the input stream contents

the frames that compose it) determines the input-output relation for the video decoder. Because the size of frames in the MPEG2 video stream is variable, and the size of packets is fixed, this makes that a variable number of encoded frames will be contained in 1 FP on some segments of the input stream, and a variable number of FPs will contain one encoded frame on other segments of the stream. This implies that $x$ is variable and moreover, the input-output relation can change from $1:x$ to $x:1$ and vice-versa. In a further section we illustrate this kind of behaviour in Figure 6.8 and Figure 6.9 where we present snapshots of a system execution in practice, on a VLIW TriMedia processor. We present below (Figure 6.1) the program and associated trace set for the video decoder which in our case study is the second component in the chain ($C_2$).

At the programming level we add the following basic statements:

- $\text{seek\_next\_end\_frame}(p,\text{end\_frame})$ that seeks the first end of frame within a packet $p$ and stores it in $\text{end\_frame}$. The statement returns NULL if no frame end has been found in packet $p$.
- $\text{retrieve\_frame}(p,\text{frame})$ that stores an entire frame or part of a frame from packet $p$ in variable $\text{frame}$. In the case that a part of a frame is stored in $\text{frame}$, then the part is contiguously added to the parts already stored in $\text{frame}$.

The corresponding trace actions in the trace alphabet of the new basic statements are defined below:

$$
\text{Alph}\left(\text{seek\_next\_end\_frame}\ (\text{VAR\ LIST})\right) \overset{def}{=} \left\{ \text{sne}\ (\text{VAR\ LIST})^1, \ldots, \text{sne}\ (\text{VAR\ LIST})^{m_{21}} \right\}
$$

$$
\text{Alph}\left(\text{retrieve\_frame}\ (\text{VAR\ LIST})\right) \overset{def}{=} \left\{ \text{rf}\ (\text{VAR\ LIST})^1, \ldots, \text{rf}\ (\text{VAR\ LIST})^{m_{22}} \right\}
$$

$$
\text{Tr}\left(\text{seek\_next\_end\_frame}\ (\text{VAR\ LIST})\right) \overset{def}{=} \not\exists \text{sne}\ (\text{VAR\ LIST})^1 \cdots \text{sne}\ (\text{VAR\ LIST})^{m_{21}} \not\exists
$$

$$
\text{Tr}\left(\text{retrieve\_frame}\ (\text{VAR\ LIST})\right) \overset{def}{=} \not\exists \text{rf}\ (\text{VAR\ LIST})^1 \cdots \text{rf}\ (\text{VAR\ LIST})^{m_{22}} \not\exists
$$
$C_2$: frame := NULL; end_frame := NULL;
receive(fq1, p);
seek_next_end_frame(p, end_frame);
while(true) do
{
    // Begin "x:1" loop. Multiple packets contain 1 frame.
    while(end_frame == NULL) do
    {
        retrieve_frame(p, frame);
        send(bq1, p);
        receive(fq1, p);
        seek_next_end_frame(p, end_frame);
    } // End of frame found.
    retrieve_frame(p, frame);
    // Retrieved, stored 1 entire frame in variable "frame".
    receive(bq2, q); process_fct2(frame, q); send(fq2, q);
    // Frame decoded and sent to $C_3$.
    seek_next_end_frame(p, end_frame);

    // Begin "1:x" loop. One packet contains multiple frames.
    while(end_frame! = NULL) do
    {
        retrieve_frame(p, frame);
        receive(bq2, q); process_fct2(frame, q); send(fq2, q);
        // Frame decoded and sent to $C_3$.
        seek_next_end_frame(p, end_frame);
    } // End of packet reached.
}.

Figure 6.1. Program of a component with execution dependent on the contents of the input stream.
The traceset is
\[ \text{Tr}(C_2) = \{ \text{\textit{frame}} := \text{NULL} \land \text{end\_frame} := \text{NULL} \land \text{fq}_1 \land \ldots \land \text{snef}(p, \text{end\_frame})^{m_1} \land t^{e_0} \mid t \in \text{Tr}(C'_2) \} \]
where
\[ \text{Tr}(C'_2) = \{ \text{\textit{C}}^{e_1}_{c_2} \land \text{t}_{md}^{l_1} \land \text{t}_{md}^{l_2} \land \text{t}_{md}^{l_3} \land \text{t}_{md}^{l_4} \mid p^{l_1} \in \text{Tr}(C'_2) \land p^{l_2} \in \text{Tr}(C'_2) \land p^{l_3} \in \text{Tr}(C'_2) \land p^{l_4} \in \text{Tr}(C'_2) \} \]
and
\[ \text{Tr}(C_{2x:1}) = \{ \text{\textit{end\_frame}}! = \text{NULL} \land \text{x}, \ldots \land \text{snef}(p, \text{end\_frame})^{m_1} \land \text{rf}(p, \text{frame})^{m_2} \land \text{fq}_2! \land \text{snf}(p, \text{end\_frame})^{m_1} \land \text{rf}(p, \text{frame})^{m_2} \} \]
\[ \text{Tr}(C_{2x:2}) = \{ \text{\textit{end\_frame}}! = \text{NULL} \land \text{x}, \ldots \land \text{snf}(p, \text{end\_frame})^{m_1} \land \text{rf}(p, \text{frame})^{m_2} \land \text{snf}(p, \text{end\_frame})^{m_1} \land \text{rf}(p, \text{frame})^{m_2} \} \].

Note also that relations (2.4) and (2.5) that describe the flow and recycling of packets hold for this component as well. Indeed the program and traceset of the component show that each input full packet is recycled before a new full packet is received and each empty packet is sent into the corresponding output forward queue before a new empty packet is received.

Also we add two more assumptions that express the relation between the deferral times of a component with deferred execution and the computation time of of the snef and rf additional processing actions of C2:

\[ 2 \sum_{k=1}^{n} \tau_k < \sum_{j=1}^{m_1} \delta(\text{snef}(p, \text{end\_frame})^j) \]  \hspace{1cm} (6.1)  

and

\[ 2 \sum_{k=1}^{n} \tau_k < \sum_{j=1}^{m_2} \delta(\text{rf}(p, \text{frame})^j). \]  \hspace{1cm} (6.2)  
The process of selecting the system trace is identical with the one presented in Chapter 4. At the end of this process we find that there exists a unique trace $\rho$ that specifies the execution of the chain.

### 6.2 A chain without timing constraints

The system we analyze in this section is illustrated in Figure 6.2.

![Figure 6.2. Chain composed of a component with deferred execution, a component with execution dependent on the contents of the input stream, and N-2 data-driven components.](image)

**Lemma 6.1.** Let $C_i$ be such that $(\forall j : j < i : P(C_j) > P(C_i))$. Then there does exist a state $t$ of $\rho$ such that in any state $s (t \subseteq s \subseteq b_{q_{i-1}} \subseteq \rho), C_j b b_{q_j} \in s$ of $T_{cc}$ for all $j < i$.

**Proof.** Identical with proof of Lemma 5.1 given that (2.4) and (2.5) hold for $C_2$ in this case as well.

Given that (2.4) and (2.5) hold for $C_2$ in this case as well, the statement given in Lemma 3.2 holds here again:

Let $C_i$ be such that $(\forall j : j > i : P(C_j) > P(C_i))$ and consider a state $s$ of $\rho$ such that the next action after $s$ in $\rho$ is one of $A(C_i)$. Then $C_j b b_{q_{j-1}} \in s$ of $T_{cc}$ for all $j > i$.

Both lemmas above hold for an overall minimum in priority $P(C_m)$. We make use of this property of the minimum in priority in the following theorem which presents the execution of the system as two parameterized patterns (with parameter $x$) of chain execution corresponding to the two behaviours of $C_2$ according to its $x : 1$ and $1 : x$ input-output relations. Given the fact that this system is theoretical and the results of this analysis are used as an intermediary step towards the system analyzed in the next section, we restrict the case analysis to $P(C_2) < P(C_1)$ if $m > 2$. 
6.2 A chain without timing constraints 141

Theorem 6.1. Consider a system as in Figure 6.2, where $P(C_2) < P(C_1)$ if $m > 2$. After a finite prefix $t_{init}$, the trace $\rho$ recording the execution of the chain will adopt one of the 2 parameterized patterns (with parameter $x$) of chain execution corresponding to the two behaviours of $C_2$ according to its $x:1$ and $1:x$ input-output relations:

$$\rho = t_{init} \sim ((s_{x:1})^l \sim (s_{1:x})^k)^0$$

where:

A. $x:1$ input-output relation

$$s_{x:1} = \left\{ \begin{array}{ll}
t_{C_m} \sim t_L \sim t_m^{md} \sim t_{C_2}^{1x'} \sim t_{C_2}^{x1} \sim f_{q_m} \sim t_R & \text{if } m > 2 \\
t_{C_2}^{x1} \sim t_{C_2}^{md} \sim t_R & \text{if } m \leq 2 \\
\end{array} \right.$$  

- $t_{C_m} = \bar{x}$ true $f_{q_{m-1}} \sim b_{m-1} \sim \bar{c}_m \sim \ldots \sim \bar{c}_1 \sim b_1 \bar{x}$.
- $t_L$ records the interleaved execution of components $C_1, \ldots, C_{m-1}$.
- $t_R$ records the interleaved execution of components in $RSC_m$.

If $m > 2$:

- $t_{C_j}^{1x'} = \bar{x}$ end frame = NULL $\bar{x}$.
- $t_{C_2}^{x1} \sim t_{C_2}^{md} \sim t_R$ records the interleaved execution of actions in:

  - $(t_{C_2}^{x1'}) \sim r_{f(p,frame)} \sim \ldots \sim r_{f(p,frame)}^{m2}$,
  - $(t_{C_1})^x$ and
  - $\bar{x}f_{q_j} \sim \bar{x}f_{q_j-1} \bar{x}$, where $P(C_j) < P(C_1)$, $\forall j, 2 < j < m$.

- The exact order of actions in $t_{C_2}^{x1} \sim t_{C_1}^{x}$ is determined by the priorities of $C_2, C_j$, $(2 < j < m)$ and $C_1$, the computation times of actions of $C_2$ and $C_j$, $(2 < j < m)$, and the length of the deferral times of $C_1$.

B. $1:x$ input-output relation

$$s_{1:x} = \left\{ \begin{array}{ll}
s_{1:x}^0 \sim (s_{1:x})^{x-2} \sim s_{1:x}^3 & \text{if } m > 2 \\
t_{C_2}^{1x} \sim t_{C_2}^{md} \sim (t_R)^x \sim & \text{if } m \leq 2 \\
\end{array} \right.$$  

With:

$$s_{1:x}^0 = t_{C_m} \sim t_L \sim t_{C_2}^{md} \sim t_{C_2}^{1x'} \sim f_{q_m} \sim t_R.$$  

$$s_{1:x}^2 = t_{C_m} \sim t_L \sim t_{C_2}^{1x'} \sim t_{C_2}^{1x''} \sim f_{q_m} \sim t_R.$$
\[ s_{1:x}^3 = t_{c_m} \prec t_L \prec t_{c_2}\ presuperscript{1, x''} \prec t_{c_2} \prec t_{c_2 c_1} \prec f_{q_m} \prec t_R. \]

- \( t_{c_m} = \text{true} \prec f_{q_{m-1}} \prec b_{q_m} \prec c_1 \prec \ldots \prec c_{m} \prec b_{q_{m-1}} \).
- \( t_L \) records the interleaved execution of components \( C_3, \ldots, C_{m-1} \).
- \( t_R \) records the interleaved execution of components in \( RSC_m \).

If \( m > 2 \):
- \( t_{c_2}^{1, x'} \) as sub-trace of \( s_{1:x}^1 \), and \( s_{1,x}^2 \) is equal to
  \( \text{end_frame}! = \text{NULL} \prec rf(p, frame)^1 \prec \ldots \prec rf(p, frame)^{m_{22}} \).
- \( t_{c_2}^{1, x'} \) as sub-trace of \( s_{1:x}^3 \) is equal to \( \text{end_frame} = \text{NULL} \).
- \( t_{c_2 c_1} \) records the interleaved execution of actions in:
  - \( t_{c_1}^{1, x} \)
  - \( t_c \) and
  - \( \text{end_frame}! \prec \text{true} \prec f_{q_j} \prec b_{q_{j-1}} \text{ where } P(C_j) < P(C_1), \ 2 < j < m \).

- The exact order of actions in \( t_{c_2 c_1} \) is determined by the priorities of \( C_2 \) \( C_j \), \( 2 < j < m \) and \( C_1 \), the computation times of actions of \( C_2 \) and \( C_j \), \( 2 < j < m \), and the length of the deferral times of \( C_1 \).

If \( m \leq 2 \)
\[ t'_R = t_{c_2}^{1, x'} \prec b_{q_2} \prec c_1 \prec \ldots \prec c_{m} \prec f_{q_2} \prec t_R \prec snef(p, frame)^{m_{1}} \prec snef(p, frame)^{m_{1}}. \]

**Proof.** We prove the above statements by construction of trace \( \rho \).

**A.** \( x : 1 \) input-output relation

We must prove that
\[ s_{x:1} \begin{cases} t_{c_m} \prec t_L \prec t_{c_2}^{1, x''} \prec t_{c_2} \prec t_{c_2 c_1} \prec f_{q_m} \prec t_R & \text{if } m > 2 \\ t_{c_2 c_1} \prec t_{c_2}^{1, x} \prec t_R & \text{if } m \leq 2 \end{cases} \]

**1.** \( m > 2 \)

The execution of the system in this case is illustrated in Figure 6.3. At the end of \( t_{\text{init}} \) all components in \( LSC_m \) are blocked at action \( b_{q_i} \), \( 0 < i < m \) and all components in \( RSC_m \) are blocked at action \( f_{q_{i-1}} \), \( m < i < N \). The only component ready-to-run is \( C_m \). Given \( m > 2 \) follows that \( C_m \) is a data-driven component, therefore it will execute according to its program trace
\[ t_{c_m} = \text{true} \prec f_{q_{m-1}} \prec b_{q_m} \prec c_1 \prec \ldots \prec c_{m} \prec b_{q_{m-1}} \text{.} \]
6.2 A chain without timing constraints

Figure 6.3. Execution of the system according to the \( x : 1 \) pattern. Case A. where \( m > 2 \).

When action \( bq_{m-1}! \) is executed it de-blocks in cascade components in \( LSC_m \). We restrict our attention for the moment to the execution of data-driven components \( C_3, \ldots, C_{m-1} \). Before \( C_m \) executes again all these components execute one iteration of their loop after which they become blocked once more at action \( bq_i? \), \( 2 < i < m \). Their execution is interleaved and the exact order is determined by their priorities. This execution is denoted with \( \xi \). At this point we can express \( s_{x:1} \) as:

\[
 s_{x:1} = t_{C_m} \sim t_L \sim s.
\]

During \( t_L \), some of the components will have been preempted when executing action \( bq_{i-1}! \), \( 2 < i < m \) which de-blocked a predecessor neighbour with a higher priority. The rest of the actions to be executed before becoming blocked again at action \( bq_i? \) are:

\[
 \hat{\xi}fq_1! - true - \hat{\xi}fq_{i-1}? \hat{\xi}, \ 2 < i < m.
\]

Among these components, those with a lower priority than \( C_1 \), and which have not completed their loop iteration before \( C_1 \) is de-blocked, will execute interleaved with \( C_1 \). We denote this set of components with \( BC_1 \) and we will refer to this situation again later on.

When \( C_2 \) is de-blocked, it executes first trace \( t^{\text{end}}_{C_2} \) and continues with trace \( t^{1:x'}_{C_2} \) according to its program. Because this is the \( x : 1 \) case meaning that multiple packets contain one frame the test in \( t^{1:x'}_{C_2} \) returns false which makes that in this case

\[
 t^{1:x'}_{C_2} = \hat{\xi} end_frame = NULL \hat{\xi}.
\]

At this point we can express \( s_{x:1} \) as:

\[
 s_{x:1} = t_{C_m} \sim t_L \sim t^{\text{end}}_{C_2} \sim t^{1:x'}_{C_2} \sim s.
\]

The execution of \( C_2 \) continues with \( t^{1:mm}_{C_2} \), \( C_2 \) executes \( bq_1! \) which de-blocks \( C_1 \). \( C_1 \) executes one iter-
ation (recorded in \( t_{C_1} \)) after which it blocks again at action \( bq_1 \). Given that \( P(C_2) < P(C_1) \), the two components execute interleaved with each other, with \( C_2 \) making use of the deferral times of \( C_1 \). Also given (6.1), follows that during each iteration of \( C_1 \) the deferral times of \( C_1 \) are completely consumed. Due to the \( x : 1 \) input-output relation of \( C_2 \), \( t_{C_2}^{x:1} \) is executed \( x \) times, each time de-blocking \( C_1 \) to execute one iteration of its loop. Coming back to the components in \( BC_{C_1} \), given the fact that they have a lower priority than \( C_1 \) they will also execute interleaved with \( C_1 \), possibly making use of the deferral times of \( C_1 \) (depending on the duration of the deferral times, the computation times of component actions and the priority assignment). The actions executed by \( C_j \in BC_{C_1} \) are

\[
t_{C_j} = \frac{x}{2}f_{q_1} \rightarrow \text{true} \rightarrow f_{q_i-1}?, \quad 2 < i < m.
\]

After \( x \) iterations of \( t_{C_2}^{x:1} \), \( C_2 \) completes \( t_{C_2}^{x:1} \) with actions \( rf(p, \text{frame})^1 \rightarrow \ldots \rightarrow rf(p, \text{frame})^{m2} \) according to its program. The interleaved execution of \( C_2 \), \( C_1 \) and \( C_j \in BC_{C_1} \) is denoted with \( t_{C_2C_jC_1}^{x:1} \).

Therefore we can express \( s_{x:1} \) as:

\[
s_{x:1} = t_{C_m} \rightarrow t_L \rightarrow t_{md}^{C_2} \rightarrow t_{C_2}^{x:1} \rightarrow t_{C_jC_1}^{x:1} \rightarrow s.
\]

At the end of \( t_{C_2}^{x:1} \) \( C_2 \) becomes blocked at action \( bq_2 \) again and \( C_1 \) is blocked at \( bq_1 \). Therefore in this state all components in \( LSC_m \) are blocked at action \( bq_i \), \( 0 < i < m \). The only component \( \text{ready-to-run} \) is again \( C_m \).

\( C_m \) executes \( f_{q_m} \) according to its program which de-blocks in cascade the components in \( RSC_m \). All components execute one iteration of their loop after which they become blocked again at action \( f_{q_{i-1}} \), \( m < i < N \). The interleaved execution of components in \( RSC_m \) is recorded by trace \( t_{R} \). Therefore \( s_{x:1} \) can be expressed as

\[
s_{x:1} = t_{C_m} \rightarrow t_L \rightarrow t_{md}^{C_2} \rightarrow t_{C_2}^{x:1} \rightarrow t_{C_jC_1}^{x:1} \rightarrow f_{q_m} \rightarrow t_{R}.
\]

2. \( m \leq 2 \)

To be proven in the same manner shown above.

Given that at the end of \( s_{x:1} \) all components in the chain except \( C_m \) are blocked, the situation is identical with the one at the end of \( t_{\text{init}} \). This means that the execution according to the \( x : 1 \) pattern is repetitive and continues as long as the input full packet in front of \( C_2 \) contains part of a frame. This implies that \( \rho \) can be expressed as

\[
\rho = t_{\text{init}} \rightarrow (s_{x:1})^l \rightarrow s.
\]

B. \( 1 : x \) input-output relation
6.2 A chain without timing constraints

We must prove that

\[ s_{1:x} = \begin{cases} 
  s_{1:x}^1 \sim (s_{1:x}^2)^{x-2} \sim s_{1:x}^3 & \text{if } m > 2 \\
  t_{C_2C_1}^{md} \sim (t_R)^{x-1} & \text{if } m \leq 2 
\end{cases} \]

With:

- \( s_{1:x}^1 = t_{C_m} - t_L - t_{C_2}^{md} - t_{C_2}^{x'} - f_{q_m} \sim t_R \)
- \( s_{1:x}^2 = t_{C_m} - t_L - t_{C_2}^{x''} - t_{C_2}^{x'} - f_{q_m} \sim t_R \)
- \( s_{1:x}^3 = t_{C_m} - t_L - t_{C_2}^{x''} - t_{C_2}^{x'} - t_{C_2C_1}^{x''} - f_{q_m} \sim t_R \)

1. \(- m > 2 \)

The execution of the system in this case is illustrated in Figure 6.4.

---

**Figure 6.4.** Execution of the system according to the 1 : x pattern. Case B. where \( m > 2 \).

At the end of \( t_{\text{init}} \) all components in \( LSC_m \) are blocked at action \( bq_i \), \( 0 < i < m \) and all components in \( RSC_m \) are blocked at action \( f_{q_{m-1}} \), \( m < i < N \). The only component \( \text{ready-to-run} \) is \( C_m \). Given \( m > 2 \) follows that \( C_m \) is a data-driven component, therefore it will execute according to its program trace

\[ t_{C_m} = \hat{x} f_{q_{m-1}} \sim bq_{m-1} \sim c_{m} \sim \ldots \sim c_{m} \sim bq_{m-1} ! \hat{x} \]

When action \( bq_{m-1} ! \) is executed it de-blocks in cascade components in \( LSC_m \). Before \( C_m \) executes again all data-driven components \( C_3, \ldots, C_{m-1} \) execute one iteration of their loop after which they become blocked once more at action \( bq_i \), \( 2 < i < m \). Their execution is interleaved and the exact order is determined by their priorities. This execution is denoted with \( \hat{x} \). At this point we can express \( s_{1:x} \) as:

\[ s_{1:x} = t_{C_m} \sim t_L \sim s \]

When \( C_2 \) is de-blocked, it executes first trace \( t_{C_2}^{md} \) and continues with trace \( t_{C_2}^{x'} \) according to its program. Because this is the 1 : x case meaning that multiple frames are contained in one packet, the test in \( t_{C_2}^{x'} \) returns true which makes that in this case
\( t_{C_2}^{1:x} = \texttt{end_frame!} = \text{NULL} \quad \text{rf(p, frame)}^1 \quad \ldots \quad \text{rf(p, frame)}^{m_{22}} \).

So far we can express \( s_{1:x} \) as:

\[
s_{1:x} = t_{C_m} \quad t_L \quad t_{C_2}^{1:x} \quad s.
\]

At this point the execution of \( C_2 \) would continue with \( t_{C_2}^{1:x''} \) according to its program, however because the first action of \( t_{C_2}^{1:x''} \) is \( bq_2 \)?, \( C_2 \) becomes blocked (the empty packet produced by \( C_3 \) was consumed and \( C_3 \) did not produce another empty packet ever since).

Notice that in this state \( C_1 \) is still blocked at action \( bq_1 \) as it was in state \( t_{\text{init}} \) because \( C_2 \) has not executed \( bq_1 \) so far. Therefore in this state all components in \( LSC_m \) are blocked at action \( bq_i \), \( 0 < i < m \). The only component \( \text{ready-to-run} \) is again \( C_m \).

\( C_m \) executes \( fq_m! \) according to its program which de-blocks in cascade the components in \( RSC_m \). All components execute one iteration of their loop after which they become blocked again at action \( fq_{i-1} \), \( m < i < N \). The interleaved execution of components in \( RSC_m \) is recorded by trace \( t_R \). Therefore so far \( s_{1:x} \) can be expressed as

\[
s_{1:x} = t_{C_m} \quad t_L \quad t_{C_2}^{1:x} \quad fq_m! \quad t_R \quad s.
\]

or as

\[
s_{1:x} = s_{1:x}^1 \quad s.
\]

Because all components in the chain (except \( C_m \)) are blocked, \( C_m \) executes next. Its execution is recorded by \( \kappa_m \). As we have seen above, \( t_{C_m} \) is followed by \( t_L \) that records the interleaved execution of \( \text{data-driven components} \) \( C_3, \ldots, C_{m-1} \).

When \( C_2 \) is de-blocked this time, it continues with \( t_{C_2}^{1:x''} \) and \( t_{C_2}^{1:x'} \) according to its program. At the end of \( t_{C_2}^{1:x'} \) \( C_2 \) becomes blocked again at action \( bq_2 \)?.

The only component \( \text{ready-to-run} \) is again \( C_m \). \( C_m \) executes \( fq_m! \) according to its program which de-blocks in cascade the components in \( RSC_m \). The interleaved execution of components in \( RSC_m \) is recorded by trace \( t_R \).

So far \( s_{1:x} \) can be expressed as

\[
s_{1:x} = s_{1:x}^1 \quad t_{C_m} \quad t_L \quad t_{C_2}^{1:x'} \quad fq_m! \quad t_R \quad s.
\]

or as

\[
s_{1:x} = s_{1:x}^1 \quad s_{1:x}^2 \quad s.
\]

The execution specified by \( s_{1:x}^2 \) will repeat \( x-2 \) times in the case of a packet that contains \( x \) frames (one frame has been already produced during \( s_{1:x}^1 \)). Each time during \( s_{1:x}^2 \) the test \( \text{end_frame!} = \text{NULL} \) returns \( \text{true} \) therefore during \( s_{1:x}^2 \) \( t_{C_2}^{1:x'} \) is:
6.2 A chain without timing constraints

\[ t^{1:x'}_{C_2} = \text{end_frame}! = \text{NULL} \sim r f(p, frame)^{1} \sim \ldots \sim r f(p, frame)^{m22}. \]

\[ s \sim s_{1:x} \text{ can be expressed as } s_{1:x} = s_{1:x} \sim (s_{1:x}^{2})^{x-2} \sim s. \]

At the end of \( s_{1:x}^{2} \sim x-2 \) all but the last frame have been produced in \( f_{C_2} \) and \( C_2 \) all components in the chain except \( C_m \) are blocked again. Therefore the next component to execute is \( C_m \) (execution recorded by \( t_{C_m} \)). \( t_{C_m} \) is followed by \( t_L \) that records the interleaved execution of data-driven components \( C_3, \ldots, C_{m-1} \).

Some components have been preempted and do not finish their loop iteration before \( C_1 \) starts executing \((C_j \in BC_{C_1}, P(C_j) < P(C_1))\).

When \( C_2 \) is de-blocked it executes \( t^{1:x'}_{C_2} \) (during which it produces the last frame) followed by \( t^{1:x'}_{C_2} \) according to its program. Since the end of packet has been reached, the test in \( t^{1:x'}_{C_2} \) returns \text{true} meaning that

\[ t^{1:x'}_{C_2} = \text{end_frame} = \text{NULL}. \]

Next \( C_2 \) executes \( t^{1:x'}_{C_2} \) according to its program. During \( t^{1:x'}_{C_2} \) action \( b_q! \) is executed which de-blocks component \( C_1 \). From here on \( C_2, C_1 \) and \( C_j \in BC_{C_1}, P(C_j) < P(C_1) \) execute interleaved. This interleaved execution is recorded in trace \( t^{1:x'}_{C_2}C_1C_2C_1 \).

At the end of \( t^{1:x'}_{C_2}C_1C_2C_1 \) all components are blocked except \( C_m \). When \( C_m \) executes \( f_{C_m} \) it de-blocks in cascade components in \( RSC_m \). Their interleaved execution is recorded by \( t_R \). At this point \( s_{1:x} \) can be expressed as

\[ s_{1:x} = s_{1:x} \sim (s_{1:x}^{2})^{x-2} \sim t_{C_m} \sim t_L \sim t^{1:x'}_{C_2} \sim t^{1:x'}_{C_2} \sim t^{1:x'}_{C_2} \sim f_{C_m} \sim t_R \sim s. \]

or as

\[ s_{1:x} = s_{1:x} \sim (s_{1:x}^{2})^{x-2} \sim s_{1:x}^{3}. \]

At the end of \( s_{1:x}^{3} \) all components are blocked again except \( C_m \).

\[ 2. - m \leq 2 \]

To be proven in the same manner shown above.

Given that at the end of \( s_{1:x} \) all components in the chain except \( C_m \) are blocked, the situation is identical with the one in the state preceding \( s_{1:x} \). This means that the execution according to the \( 1 : x \) pattern is repetitive and continues as long as the input full packet in front of \( C_2 \) contains multiple frames.

Notice that both at the end of \( s_{1:x} \) as well as at the end of \( s_{1:x} \), during trace \( t^{1:x}_{C_2} \) the decision about continuing with the current pattern or to change to the other one is taken. That is because during trace \( t^{1:x}_{C_2} \) a new full packet is received as input, and the end of the first frame is sought within that packet, therefore determining whether the next execution of \( C_2 \) is going to be according to pattern \( 1 : x \) or \( x : 1 \).
This implies that under the assumption of infinite input stream \( \rho \) can be expressed as
\[
\rho = t_{\text{init}} - (s_{x:1})^l - (s_{1:x})^k.
\]

Note that
\[
\#(s_{1:x} \uparrow A(C_1), a) = 1, \forall a \in A(C_1).
\]
\[
\#(s_{1:x} \uparrow A(C_i), a) = x, \forall a \in A(C_i), 1 < i \leq N.
\]
\[
\#(s_{x:1} \uparrow A(C_i), a) = 1, \forall a \in A(C_i), 1 < i \leq N.
\]

6.3 A chain with timing constraints

The system we analyze in this section is illustrated in Figure 6.5. Theorem 6.1 shows that one full packet is sent in \( f_{q_{N-1}} \) at the end of each of the following traces: \( s_{x:1}, s_{1:x}^1, s_{1:x}^2 \), and \( s_{1:x}^3 \). We consider
\[
S^M = \max(S^M_0(s_{x:1}), S^M_0(s_{1:x}^1), S^M_0(s_{1:x}^2), S^M_0(s_{1:x}^3))
\]
where \( S^M_0(s_{x:1}), S^M_0(s_{1:x}^1), S^M_0(s_{1:x}^2), S^M_0(s_{1:x}^3) \) are sums of worst case computation times of the actions that compose the traces in the given input stream.

The lemma below shows that when \( S^M < 2T_N \) there exists a state in which all forward queues are filled to their capacities, which implies all backward queues drained.

**Lemma 6.2.** \( S^M < 2T_N \Rightarrow \exists t_{\text{init}} \in St(\rho): C_i \ b_{q_i}? [\text{in } t_{\text{init}} \text{ of } T_{cc}], \forall i, 1 \leq i < N \)

**Proof.** Lemma 6.1 shows that there exists a state where all forward queues
preceeding $C_m$ are filled to their capacity (backward queues drained). Similar approach used for Lemma 4.1 to show the filling up of the rest of the forward queues in the system.

**Corollary 6.1.** $S^M < 2T_N \land P(C_N) = \max_{i=1}^N P(C_i) \Rightarrow$

\[
SQoSci(t_{\text{init}}, \varepsilon, f_{q_{N-1}}, f_{q_N}, f_{q_N}!, b_{q_N}, f_{q_N}!, T_0^N, T_N).
\]

**Proof.** Identical with the one of Corollary 4.5

The theorem below states that given $S^M < 2T_N$, after a finite prefix $t_{\text{init}}$ of trace $\rho$ the system executes according to a parameterized pattern of execution.

**Theorem 6.2.** Consider a system as illustrated in Figure 6.5. Given that $S^M < 2T_N$ holds, after a finite prefix $t_{\text{init}}$, the trace $\rho$ recording the execution of the chain will adopt one of the 2 parameterized patterns (with parameter $x$) of chain execution corresponding to the two behaviours of $C_2$ according to its $x : 1$ and $1 : x$ input-output relations. Depending on the nature of the input stream $\rho$ can be:

\[
\rho = t_{\text{init}} \rightarrow \rho_{\text{stable}}
\]

where

\[
\rho_{\text{stable}} = \left((s_{1:x})^p \rightarrow (s_{x:1})^k\right)_{x_0}
\]

or

\[
\rho_{\text{stable}} = \left((s_{x:1})^p \rightarrow (s_{x:1})^k\right)_{x_0}
\]

Sub-traces $s_{x:1}$ and $s_{1:x}$ specify the execution of the chain processing portions of the stream that determine an $x : 1$ and respectively a $1 : x$ input-output relation for $C_2$:

**A. $x : 1$ input-output relation**

\[
s_{x:1} = t_{C_1}^1 \sim t_{LSC}^1 \sim d(T_0^N + i \ast T_N) \sim t_{C_2}^2 \sim t_{LSC}^2 \sim d(T_0^N + i \ast T_N)
\]

- $t_{LSC}^1$ and $t_{LSC}^2$ are sub-traces of $t_{LSC}$ and $t_{LSC}^1 \sim t_{LSC}^2 = t_{LSC}^1$ where

\[
t_{LSC} = t_{L} \sim t_{C_2}^1 \sim t_{LSC}^1 \sim t_{C_2}^1 \sim t_{C_2}^1 \sim \ldots \sim t_{C_2}^1 \sim t_{LSC}^1
\]

- $t_L$ records the interleaved execution of components $C_3, \ldots, C_{N-1}$.

- $t_{C_2}^1 = \xi \end_frame = \text{NULL} \xi$.

- $t_{C_2}^1 \sim \xi \end_frame = \text{NULL} \xi$.

- $t_{C_2}^1 \sim \xi \end_frame = \text{NULL} \xi$.

**B. $1 : x$ input-output relation**

\[
s_{1:x} = t_{C_1}^1 \sim t_{LSC}^1 \sim d(T_0^N + i \ast T_N) \sim t_{C_2}^2 \sim t_{LSC}^2 \sim d(T_0^N + i \ast T_N)
\]

- $t_{LSC}^1$ and $t_{LSC}^2$ are sub-traces of $t_{LSC}$ and $t_{LSC}^1 \sim t_{LSC}^2 = t_{LSC}^1$ where

\[
t_{LSC} = t_{L} \sim t_{C_2}^1 \sim t_{LSC}^1 \sim t_{C_2}^1 \sim t_{LSC}^1 \sim \ldots \sim t_{C_2}^1 \sim t_{LSC}^1
\]

- $t_L$ records the interleaved execution of components $C_3, \ldots, C_{N-1}$.

- $t_{C_2}^1 \sim \xi \end_frame = \text{NULL} \xi$.

- $t_{C_2}^1 \sim \xi \end_frame = \text{NULL} \xi$.

- $t_{C_2}^1 \sim \xi \end_frame = \text{NULL} \xi$.
The exact order of actions in $t_{C_2C_iC_1}^{1}$ is determined by the priorities of $C_2 C_j$, $(2 < j < m)$ and $C_1$, the computation times of actions of $C_2$ and $C_j$, $(2 < j < m)$, and the length of the deferral times of $C_1$.

At the end of $s_{C_1}$ all components in $LSC_N$ are blocked at $bq_i^1$, $1 \leq i < N$.

**B. 1 : x input-output relation**

$$s_{1x} = s_{1x}^1 - (s_{1x}^2)^{i-2} - s_{1x}^3$$

With:

$$s_{1x} = t_{C_1}^1 - t_{LSC_N}^1 - d(T_0^N + i \cdot T_N) - t_{C_1}^2 - t_{LSC_N}^2 - d(T_0^N + i \cdot T_N).$$

$$s_{2x} = t_{C_1}^1 - t_{LSC_N}^2 - d(T_0^N + i \cdot T_N) - t_{C_1}^2 - t_{LSC_N}^2 - d(T_0^N + i \cdot T_N).$$

$$s_{3x} = t_{C_1}^1 - t_{LSC_N}^3 - d(T_0^N + i \cdot T_N) - t_{C_1}^2 - t_{LSC_N}^2 - d(T_0^N + i \cdot T_N).$$

- $t_{LSC_N}^{1'}$ and $t_{LSC_N}^{1''}$ are sub-traces of $t_{LSC_N}^1$.
- $t_{LSC_N}^{2'}$ and $t_{LSC_N}^{2''}$ are sub-traces of $t_{LSC_N}^2$.
- $t_{LSC_N}^{3'}$ and $t_{LSC_N}^{3''}$ are sub-traces of $t_{LSC_N}^3$.

$t_L$ records the interleaved execution of components $C_3, \ldots, C_{N-1}$.

$t_{C_2}^{1'}$ as sub-trace of $s_{1x}^1$ and $s_{1x}^2$ is equal to

$$\_\text{end}\_\text{frame}^! = NULL - rf(p, frame)^x.$$

$t_{C_2}^{1''}$ as sub-trace of $s_{1x}^1$ is equal to $\_\text{end}\_\text{frame} = NULL^x$.

$t_{C_2C_iC_1}^{1}$ records the interleaved execution of actions in:

- $t_{C_2}^{1}$,
- $\_\text{if}\_q_j^! - true^ - \_\text{if}\_q_{j-1}^? x^$ where $P(C_j) < P(C_1)$, $2 < j < N$ and
- $t_{C_1}$.

The exact order of actions in $t_{C_2C_iC_1}^{1}$ is determined by the priorities of $C_2$ and $C_1$, the computation times of actions in $t_{C_2C_1}^{1}$ and $\_\text{if}\_q_j^! - true^ - \_\text{if}\_q_{j-1}^? x$, $2 < j < N$, and the length of the deferral times of $C_1$. When $P(C_2) < P(C_1)$, actions of $t_{C_2C_1}^{1}$ execute (also) during the deferral times of $C_1$. 

6.3 A chain with timing constraints

- At the end of $s_{1:x}$ all components in $LSC_N$ are blocked at $bq_i$?, $1 \leq i < N$.

Also, we have that $SQoSci(t_{stable}, t_{init}, fq_{N-1}?, f_q{N-1}, bq_N?, bq_{N}, T_0^N, T_N)$ holds as well.

**Proof.** We prove the statement of the theorem by construction of trace $\rho$. According to Lemma 6.2 after a finite prefix $t_{init}$ all backward queues $bq_i$, $1 \leq i < N$ are drained and all components $C_i$, $1 \leq i < N$ are blocked at action $bq_i$?. The only component ready-to-run is $C_N$. Therefore $C_N$ executes $t_{1:C_N}$ according to its program. So far trace $\rho$ can be expressed as

$$\rho = t_{init} \sim t_{1:C_N} \sim u.$$  

During $t_{1:C_N}$ action $bq_{N-1}$! is executed which de-blocks component $C_{N-1}$. Corollary 6.1 and $SM < 2T_N$ imply that $\sigma(t_{init} \sim t_{1:C_N}) < T_0^N + l * T_N + flush$. Hence $\sigma(t_{init} \sim t_{1:C_N}) < T_0^N + (l + 1) * T_N$ From here follows that the delay action following $t_{1:C_N}$ is postponed until no other regular actions are ready or $\sigma(t_{init} \sim t_{1:C_N}) \geq T_0^N + (l + 1) * T_N$. When $C_N$ executes $bq_{N-1}$! it causes a de-blocking in cascade of components in $LSC_N$.

**A. x : 1 input-output relation**

The execution of the system in this case is illustrated in Figure 6.6. In the case

$$x = 5$$

$$x = 3$$

$$x = 4$$

Figure 6.6. Case A. Execution of the system with timing constraints according to the $x : 1$ pattern.

$x : 1$ the execution of components in $LSC_N$ is recorded by trace

$$t_{LSC_N} = t_L \sim t_{md}^{C_2} \sim t_{lx}^{C_2} \sim t_{lx}^{C_1}.$$  

We distinguish here the following cases:
1. \( P(C_2) < P(C_1) \)

Given the fact that at the end of \( t_{\text{init}} \) all components in \( LSC_N \) are blocked at action \( bq_2 \). \( 0 < i < N \), the execution of components in \( LSC_N \) is identical to the one specified in Theorem 6.1 for the case \( m = N \). That is \( t_{C_2C_1}^{x_1m} \) records the interleaved execution of actions in:

- \( (t_{C_2}^{x_1m})^x \sim rf(p, \text{frame})^1 \cdots \sim rf(p, \text{frame})^{m_{22}} \),
- \( (t_{C_1})^x \) and
- \( \triangleright f_{q_j} \sim \text{true} f_{q_{j-1}} \triangleright \) where \( P(C_j) < P(C_1), \ 2 < j < m \).

where \( C_2 \) executes during the deferral times of \( C_1 \).

2. \( P(C_2) > P(C_1) \)

During trace \( t_{C_2C_1}^{x_1m} \), component \( C_2 \) executes \( (t_{C_2}^{x_1m})^x \) during which it produces \( x \) empty packets in \( bq_1 \).

a. \( \text{Cap}(f_{q_1}) > x \)

Although \( C_1 \) is de-blocked, it cannot execute due to \( P(C_2) > P(C_1) \). \( C_2 \) executes

\( (t_{C_2}^{x_1m})^x \sim rf(p, \text{frame})^1 \cdots \sim rf(p, \text{frame})^{m_{22}} \)

after which it becomes blocked at action \( bq_2 \). At this point \( C_1 \) executes \( (t_{C_1})^x \). Note that in this case \( C_2 \) does not execute during the deferral times.

The deferral times are used by components \( C_j \) (\( P(C_j) < P(C_1), \ 2 < j < m \)) if \( P(C_j) < P(C_2), \ 2 < j < m \).

b. \( \text{Cap}(f_{q_1}) < x \)

Again, although \( C_1 \) is de-blocked, it cannot execute due to \( P(C_2) > P(C_1) \). In this case \( C_2 \) executes \( (t_{C_2}^{x_1m})^{\text{Cap}(f_{q_1})} \) after which it becomes blocked at action \( f_{q_1} \). At this point \( C_1 \) can execute \( t_{C_1} \) during which it produces 1 full packet in \( f_{q_1} \) and de-blocks \( C_2 \). From this point on \( C_2 \) and \( C_1 \) execute alternately de-blocking each other at the input forward queue (for \( C_2 \)) and the input backward queue (in the case of \( C_1 \)) until the \( x \) iterations of \( t_{C_2}^{x_1m} \) are completed. Again, the deferral times are used by components \( C_j \) (\( P(C_j) < P(C_1), \ 2 < j < m \)) if \( P(C_j) < P(C_2), \ 2 < j < m \).

When \( T_N < s^M < 2T_N \), the execution of \( t_{LSC_N} \) is interrupted at moment \( T_0^N + (l + 1) \ast T_N + \mu \) by the delay action \( d(T_0^N + (l + 1) \ast T_N) \) which is followed by \( t_{C_2}^2 \). At the end of \( t_{C_2}^2 \), trace \( t_{LSC_N} \) is resumed and continues until its end. The two sub-traces of \( t_{LSC_N} \) are denoted by \( t'_{LSC_N} \) and \( t''_{LSC_N} \). Trace \( \rho \) can be expressed now as:

\( \rho = t_{\text{init}} \sim t_{C_1}^1 \sim t'_{LSC_N} \sim d(T_0^N + (l + 1) \ast T_N) \sim t_{C_2}^2 \sim t''_{LSC_N} \sim u. \)

Because \( s^M < 2T_N \) follows that
6.3 A chain with timing constraints

\[ \sigma(t_{\text{init}} \sim t_{C_N}^{1} \sim t_{LSC_N}^{1} \sim d(T_{0}^{N} + (l + 1) \ast T_{N}) \sim t_{C_N}^{2} \sim t_{LSC_N}^{2} < T_{0}^{N} + (l + 2) \ast T_{N} \]

Given that at the end of \( t_{LSC_N} \) all components in \( LSC_N \) are blocked again at action \( b_{qi} \), 0 < i < N implies that \( t_{LSC_N}^{2} \) is followed by the delay action \( d(T_{0}^{N} + (l + 2) \ast T_{N}) \) which advances time until moment \( T_{0}^{N} + (l + 2) \ast T_{N} \).

\[ \rho = t_{\text{init}} \sim t_{C_N}^{1} \sim t_{LSC_N}^{1} \sim d(T_{0}^{N} + (l + 1) \ast T_{N}) \sim t_{C_N}^{2} \sim t_{LSC_N}^{2} \sim d(T_{0}^{N} + (l + 2) \ast T_{N}) \sim u. \]

At moment \( T_{0}^{N} + (l + 2) \ast T_{N} \), again the only component ready-to-run is \( C_{N} \), a situation identical as at the end of \( t_{\text{init}} \) which implies repetitive execution according to this pattern as long as the input full packet in front of \( C_2 \) contains only part of a frame.

Hence \( \rho \) can be expressed now as

\[ \rho = t_{\text{init}} \sim (s_{x:1})^{p} \sim u. \]

The \( S^M < T_{N} \) case is to be handled in the similar manner.

**B. 1 : x input-output relation**

The execution of the system in this case is illustrated in Figure 6.7. In the case

\[ (s_{x:1})^{f} \]

\[ (s_{x:1})^{k} \]

\[ x = 6 \]

\[ x = 3 \]

\[ x = 4 \]

\[ 2T_{N} \]

\[ \ldots \]

\[ s_{1:x}^{1} \]

\[ s_{1:x}^{2} \]

\[ s_{1:x}^{3} \]

Figure 6.7. Case B. Execution of the system with timing constraints according to the 1 : x pattern.

1 : x the execution of components in \( LSC_N \) is recorded by trace

\[ t_{LSC_N}^{1} = t_{L} \sim t_{C_2}^{1} \sim t_{C_2}^{1 : x}. \]

We consider again \( T_{N} < S^M < 2T_{N} \). By using the same reasoning as above we find that after \( 2T_{N} \)

\[ \rho = t_{\text{init}} \sim (s_{x:1})^{p} \sim t_{C_N}^{1} \sim t_{LSC_N}^{1} \sim d(T_{0}^{N} + (l + 1) \ast T_{N}) \sim t_{C_N}^{2} \sim t_{LSC_N}^{2} \sim d(T_{0}^{N} + (l + 2) \ast T_{N}) \sim u. \]

After action \( d(T_{0}^{N} + (l + 2) \ast T_{N}) \), again the only component ready-to-run is \( C_{N} \). Next time when \( C_2 \) executes \( t_{C_N}^{1} \) and de-blocks components in \( LSC_N \) the trace recording their execution is according to Theorem 6.1

\[ t_{LSC_N}^{2} = t_{L} \sim t_{C_2}^{1} \sim t_{C_2}^{1 : x}. \]
By using the same reasoning as above we find that after $2T_N$
\[
\rho = \text{t}^{\text{init}} \sim (s_{x:1})^p \sim \\
t^1_{CN} \sim t^1_{LSNC} \sim d(T^0_N + (l + 1) * T_N) \sim t^{2}_CN \sim t^1_{LSNC} \prime \sim \\
d(T^0_N + (l + 2) * T_N) \sim \\
t^1_{CN} \sim t^2_{LSNC} \prime \sim d(T^0_N + (l + 3) * T_N) \sim t^{2}_CN \sim t^2_{LSNC} \prime \sim \\
d(T^0_N + (l + 4) * T_N) \sim \\
u.
\]

After action $d(T^0_N + (l + 4) * T_N)$, again the only component ready-to-run is $C_N$ and this execution is repeated $x - 2$ times. Therefore we have
\[
\rho = \text{t}^{\text{init}} \sim (s_{x:1})^p \sim \\
t^1_{CN} \sim t^1_{LSNC} \prime \sim d(T^0_N + (l + 1) * T_N) \sim t^{2}_CN \sim t^1_{LSNC} \prime \sim \\
d(T^0_N + (l + 2) * T_N) \sim \\
(t^1_{CN} \sim t^2_{LSNC} \prime \sim d(T^0_N + i * T_N) \sim t^{2}_CN \sim t^2_{LSNC} \prime \sim d(T^0_N + i * T_N))^{i-2} \sim \\
u.
\]

Finally when $C_N$ de-blocks in cascade components in $LSNC$, their execution is recorded by
\[
t^3_{LSNC} = t^3_{C2} \sim t^1_{C2} \sim t^2_{C2} \sim t^1_{C2} C_1.
\]

and $\rho$ becomes
\[
\rho = \text{t}^{\text{init}} \sim (s_{x:1})^p \sim \\
t^1_{CN} \sim t^1_{LSNC} \prime \sim d(T^0_N + (l + 1) * T_N) \sim t^{2}_CN \sim t^1_{LSNC} \prime \sim \\
d(T^0_N + (l + 2) * T_N) \sim \\
(t^1_{CN} \sim t^2_{LSNC} \prime \sim d(T^0_N + i * T_N) \sim t^{2}_CN \sim t^2_{LSNC} \prime \sim d(T^0_N + i * T_N))^{i-2} \sim \\
t^1_{CN} \sim t^3_{LSNC} \sim d(T^0_N + n * T_N) \sim t^{2}_CN \sim t^3_{LSNC} \sim d(T^0_N + (n + 1) * T_N) \sim \\
u.
\]

or
\[
\rho = \text{t}^{\text{init}} \sim (s_{x:1})^p \sim s^{1}_{1:x} \sim (s^{2}_{1:x})^{x-2} \sim s^3_{1:x} \sim u.
\]

which can also be expressed as
\[
\rho = \text{t}^{\text{init}} \sim (s_{x:1})^p \sim s^{1}_{1:x} \sim u.
\]

The $x$ iterations above following $(s_{x:1})^p$ in trace $\rho$ are denoted with $s_{1:x}$. Sub-trace $s_{1:x}$ is repeated as long as the input full packet in front of $C_2$ contains multiple $(x)$ frames. $\rho$ can be expressed now as
\[
\rho = \text{t}^{\text{init}} \sim (s_{x:1})^p \sim (s^{1}_{1:x})^{k} \sim u.
\]

Given that at the end of each $s_{1:x}$ the only component ready-to-run is $C_N$, a situation identical as at the end of $t^{\text{init}}$, follows that the execution according to the two patterns $1 : x$ and $x : 1$ is repetitive and infinite. Hence $\rho$ can be expressed now as
\[
\rho = \text{t}^{\text{init}} \sim (s_{x:1})^p \sim (s^{1}_{1:x})^{k} \sim u.
\]
The $S^M < T_N$ case is to be handled in the similar manner.

Obviously depending on the contents on the input full packets $\rho$ can also be

$$\rho = t_{\text{init}} \prec ((s_{1:x})^p \prec (s_{x:1})^k)^{\omega}.$$ 

Given the reasoning above and the execution of the system illustrated in Figure 6.6 and Figure 6.7 results immediately that $SQoSci((s_{x:1})^p, t_{\text{init}}, f_{q_{N-1}}?, f_{q_{N}}!, b_{q_{N}}?, f_{q_{N}}!, T_0^N, T_N)$ and $SQoSci((s_{1:x})^p, t_{\text{init}}, f_{q_{N-1}}?, f_{q_{N}}!, b_{q_{N}}?, f_{q_{N}}!, T_0^N, T_N)$ hold. Given that we have shown above for each iteration of $s_{x:1}$ and $s_{1:x}$, which implies $SQoSci(\rho, \varepsilon, f_{q_{N-1}}?, f_{q_{N}}!, b_{q_{N}}?, f_{q_{N}}!, T_0^N, T_N)$ holds as well. $\square$

**Corollary 6.2.** $(S^M < 2T_N) \land (P(C_N) = \max_{i=1}^{N} P(C_i) \Rightarrow SQoSci(\rho, \varepsilon, f_{q_{N-1}}?, f_{q_{N}}!, b_{q_{N}}?, f_{q_{N}}!, T_0^N, T_N))$

**Proof.**

corollary 6.1 gives us that under these conditions $SQoSci(t_{\text{init}}, \varepsilon, f_{q_{N-1}}?, f_{q_{N}}!, b_{q_{N}}?, f_{q_{N}}!, T_0^N, T_N)$ holds.

That $SQoSci(\rho, \varepsilon, f_{q_{N-1}}?, f_{q_{N}}!, b_{q_{N}}?, f_{q_{N}}!, T_0^N, T_N)$ holds results directly from Theorem 6.3. $\square$

### 6.4 A case study from practice - a video decoding chain

Figure 6.8 shows the recorded execution of a real system in practice on the TriMedia processor. The components after the finite prefix $t_{\text{init}}$, in a streaming chain composed of a file reader (FIRd in Figure 6.8), video decoder (VDec), a sharpness enhancement component (SSE) and a video renderer (VO). T:IDLE denotes the IDLE task in the system. In this case study the time-driven component is the video renderer VO, and $T_N$ denotes the period of the video renderer.

Note that FIRd, VDec and SSE are blocked on their input backward queues and are de-blocked in cascade by VO every $2T_N$ when VO releases 1 EP. Because of the $1:x$ input-output relation of VDec on this part of the video stream, the VDec will release only 1 EP for FIRd, which causes FIRd to be de-blocked only one time (and therefore to execute only one iteration of its trace) during $x \times 2T_N$ periods of VO.
Figure 6.8. Execution of a streaming chain composed of a File Reader, Video Decoder, Sharpness Enhancement and Video Renderer. The relation between input and output for $C_2$ (Video Decoder) is $1 : x$.

In the $x : 1$ case again (Figure 6.9) FIRd, VDec and SSE are blocked on their input backward queues and are de-blocked in cascade by VO every $2T_N$ when VO releases 1 EP.

Figure 6.9. Execution of a streaming chain composed of a File Reader, Video Decoder, Sharpness Enhancement and Video Renderer. The relation between input and output for $C_2$ (Video Decoder) is $x : 1$.

The $x : 1$ input-output relation of VDec on this part of the video stream makes it that for each FP produced by VDec (each decoded frame) VDec needs $x$ FPs from FIRd. That is why after each FP received from FIRd, VDec keeps the packet content and recycles the packet in the backward queue, which de-blocks FIRd. This process repeats $x$ times until VDec received enough FPs from FIRd to be able to produce 1 decoded frame. Given that VDec needs $x$ FPs, it will de-block FIRd $x$ times, therefore FIRd will execute $x$ iterations of its trace during each $2T_N$. 
6.5 Practical Applications

The practical application regarding optimization of memory, CPU utilization, response time and NCS are derived from the practical applications mentioned in sections 4.2 and 5.3.1. We briefly remind below the optimizations applicable for this case.

- the minimum necessary and sufficient capacity of each queue in the chain is 1 (Corollary 3.3).
- due to the same reasons explained in section 4.2.3 response time of the chain cannot be improved by assigning the minimum priority to \( C_1 \) as suggested in Chapter 3 because that does not change the influence of the time-driven component \( C_N \).
- as shown in Theorem 3.3 the response time of the chain is reduced by reducing the capacities of queues preceding \( C_N \).
- the number of context switches occurring due to the interleaved execution of the data driven components during \( t_{stable} \) can be reduced by assigning priorities as \( P(C_1) < P(C_2) < \ldots < P(C_{N-1}) \) and \( Cap(f_{qi}) = 2 \) \( \forall i, 1 \leq i < N - 1 \) (Theorem 3.2).
- Theorem 6.2 shows that the number of context switches occurring during the stable phase is lower when the system executes according to the \( 1: x \) pattern as opposed to the \( x: 1 \) pattern. Hence NCS during the stable phase can be reduced by enforcing an \( 1: x \) pattern of execution. This is done by increasing the size of packets in the queue preceding the component with behaviour dependent on the input stream content such that the input packets size is always larger than the maximum size of a frame.
- the number of context switches occurring due to interleaved execution of the data driven components with actions of \( C_N \) (during the stable phase) can only be reduced by one context switch. This is the context switch due to preemption when \( C_N \) has a higher priority than other components in the chain. This context switch can be avoided by assigning to \( C_N \) the lowest priority in the chain. However this comes at the cost of a lower \( QoS \) when \( S > T_N \).
- optimization of CPU utilization by eliminating idle times during the execution deferral is achieved by not assigning \( C_1 \) the minimum priority as shown in Property 5.2.

6.6 Summary

We have presented in this chapter a study of systems that include components with execution dependent on the contents of the input stream. Examples of
such components from the TSSA architecture are the video decoder and video encoder as well as the audio decoder and audio encoder.

We adopt an incremental approach starting from a system without timing constraints to systems with timing constraints. We show that the traces that record the execution of the systems we analyzed adopt a repetitive pattern dependent on the contents of the input stream. The two patterns correspond to the two scenarios of execution of $C_2$:

- where one full input packet contains multiple encoded frames ($1 : x$)
- where multiple full input packets contain 1 encoded frame ($x : 1$)

In the case of the system with timing constraints given that the sum of the worst computation times for one iteration of the pattern is smaller than the period length of the time-driven component, and the time-driven component has the highest priority in the chain, the quality of service requirements of the system are satisfied.

In terms of practical applications of the analysis in this chapter we show again how to optimize at design time the amount of resources (memory and CPU) used by the system, the response time of the chain, the number of context switches and the CPU utilization by minimizing the idle times.
In this chapter we study the execution of a system consisting of a linear sub-chain connected to two other linear sub-chains with timing constraints as shown in Figure 7.1. The main difference between the system studied here and those studied in all previous chapters is the system topology: in the previous chapters we have studied linear chains while in the current chapter we tackle the analysis of a system with branched topology. Aside of that, in the present case we also introduce a new type of component, the demultiplexer. In the system we study here, the sub-chain that receives input from the environment, contains a component with deferred execution (introduced in Chapter 5) and a demultiplexer component that provides input to the other two sub-chains that follow in the graph. Each of the other two sub-chains consists of a component whose execution depends on the content of the input stream, a number of data-driven components, and end with a time-driven component. The time-driven component in one of the sub-chains executes according to the interlacing standard described in section 4.2, while the other one does not. The approach we take is to decompose the system into two sub-chains $SC_v$ and $SC_o$ (Figure 7.1) that share the file reader and the demultiplexer. As a first step we characterize the execution of each sub-chain within the overall execution of the system, and subsequently we characterize the interleaving of these two executions while
Figure 7.1. A Branching Topology. The system is composed of a component with deferred execution and a demultiplexer component, that provide input to two sub-chains that follow in the graph.

pointing out what are the situations in which the QoS requirements are satisfied. In characterizing the individual execution of each sub-chain within the overall system execution we use a similar approach as presented in Chapter 6.

The chapter is organized as follows. In section 7.1 we describe the program and traceset of the demultiplexer component, as well as the invariants that specify the channel behaviour of the demultiplexer. Section 7.2 detail the assumptions we consider in our analysis while section 7.3 specifies the QoS requirement of this system. Section 7.4 presents the behavioural analysis of the system presented in this chapter. This analysis is followed by practical applications shown in section 7.5.

7.1 A demultiplexer component

In practice, the system illustrated in Figure 7.1 corresponds to a typical media streaming system in which a file reader component retrieves as input from a storage facility (a DVD for instance) a program stream containing both video and audio data. The demultiplexer component extracts the video and audio data from the input program stream and subsequently passes each data type further on as input to the two sub-chains to which it is connected in the graph. The two sub-chains receiving input from the demultiplexer have the role to decode and display the video and respectively the audio data.
7.1 A demultiplexer component

In the context of this chapter we consider as input a program stream. According to the ISO/IEC 13818-1:1996 (E) standard a program stream is created by combining one or more Packetized Elementary Streams (PES), which have a common time base, into a single stream. An elementary stream contains only one kind of data, that is either audio or video.

The program stream is designed for reasonably reliable media such as disks. In our case the program stream is retrieved from a DVD disk and comprises a number of fixed-length packs each of which is the smallest unit for transmission and multiplexing. Each pack storing DVD video or audio data contains only one packet. Also a packet stored in a program stream pack directly embodies a PES packet. We denote with $X_v$ and $X_a$ the maximum number of consecutive packets of the same type in the stream. $X_v$ corresponds to packets used as input for $f_{q_v}^2$, and $X_a$ corresponds to packets used as input for $f_{q_a}^2$. We also denote with $F_v$ and $f_v$ and respectively with $F_a$, $f_a$ the maximum and minimum number of packets that make up an encoded frame belonging to the video elementary stream and respectively to the audio elementary stream.

We present in Figure 7.2 an example of a program specifying the execution of the demultiplexer component. We use this example in the analysis of the system execution.

At the programming level we add the following basic statement:

- $determineStreamType(p, \text{streamType})$ that determines the type of the data stream stored in packet $p$. The data stream type is stored in $\text{streamType}$.

The corresponding trace actions in the trace alphabet of the new basic statements are defined below:

\[
\begin{align*}
Alph('determineStreamType( VAR LIST ') & \overset{def}{=} \\
& \{ dst( VAR LIST )^1, \ldots, dst( VAR LIST )^{m_{21}} \}
\end{align*}
\]

\[
\begin{align*}
Tr('determineStreamType( VAR LIST ') & \overset{def}{=} \\
& \{ dst( VAR LIST )^1 \sim \ldots \sim dst( VAR LIST )^{m_{21}} \}
\end{align*}
\]
While (true) do
{
    receive(fq_1, p);
    determineStreamType(p, streamType);
    if (streamType == 'video') then
    { 
        receive(bq_2, q_v);
        process(p, q_v);
        send(bq_1, p);
        send(fq_2, q_v);
    } else
    { 
        receive(bq_2, q_a);
        process(p, q_a);
        send(bq_1, p);
        send(fq_2, q_a);
    }
}

Figure 7.2. Program of a demultiplexer component.

The traceset associated with the program is:

\[ Tr(C_2) = \{ t_1 | t_1 \in Tr(C_2) \} \]

\[ Tr(C_2^0) = \{ t_\text{false}, t_\text{true} \}
\]

\[ t_2^v \triangleq dst(p, streamType)^1 \triangleright \ldots \triangleright dst(p, streamType)^{m21} \triangleright t_2 \]

\[ Tr(C_2) = \{ \text{streamType} \Rightarrow 'video' \triangleright Tr(C_2^0) \} \cup \
\{ \text{streamType} \Rightarrow 'video' \triangleright Tr(C_2) \} \]

\[ Tr(C_2^0) = \{ \text{bq}_2 \triangleright c_2(p, q_v)^1 \triangleright \ldots \triangleright c_2(p, q_v)^{m22} \triangleright bq_1 \triangleright q_v \}
\]

\[ Tr(C_2) = \{ \text{bq}_2 \triangleright c_2(p, q_v)^1 \triangleright \ldots \triangleright c_2(p, q_v)^{m22} \triangleright bq_1 \triangleright q_v \} \]

In the following we will make use of the following notations:

\[ t_{C_2}^v \triangleq f_1 \triangleright dst(p, streamType)^1 \triangleright \ldots \triangleright dst(p, streamType)^{m21} \triangleright streamType \Rightarrow 'video' \triangleright t_2^v \]
where \( t^v_2 \in Tr(C^v_2) \).

\[
t_{c^a_2} = \forall f_{q_1} ? \sim dst(p, streamType) \sim \ldots \sim dst(p, streamType)^{m_{21}} \sim \text{streamType} ! = \text{'video'} \sim t^a_2
\]

From the syntax of the demultiplexer component we obtain the following topology invariants.

\[
0 \leq \text{\#} f_{q_1} ? - \text{\#} b_{q_1} ! \leq 1 \quad (7.1)
\]

\[
0 \leq \text{\#} b_{q_2} ? - \text{\#} f_{q_2} ! \leq 1 \quad (7.2)
\]

\[
0 \leq \text{\#} b_{q_2} ? - \text{\#} f_{q_2} ! \leq 1 \quad (7.3)
\]

### 7.2 Assumptions

We decompose the branched chain we study in this chapter in two linear sub-chains \( SC_v \) and \( SC_a \) corresponding to a video and respectively audio decoding sub-chain. In practice these two sub-chains take care of video decoding and rendering and respectively the audio decoding and rendering. The two sub-chains share the file reader and the demultiplexer.

We denote with \( S^v_M \) the maximum processing time between the production of two consecutive packets in \( f_{q_1}^v \). Also the maximum processing time between the production of two consecutive packets in \( f_{q_2}^a \) is denoted by \( S^a_M \). \( S^v_M \) and \( S^a_M \) include the interfering, interleaved processing of the other sub-chain.

The assumptions about the system based on observation from practice are presented in the following. Equation (7.4) expresses that the period (denoted by \( T_M \)) of the time driven component \( C^a_M \) of \( SC_a \) is smaller than double the period (denoted by \( T_N \)) of the time driven component \( C^v_N \) of \( SC_v \), which executes according to the interlacing standard. The same equation specifies that \( T_N \) is smaller than \( T_M \).

\[
T_N < T_M < 2T_N \quad (7.4)
\]

Regarding the priority assignment to the components in the system we adopt that

\[
P(C^a_M) = \max_{C \in SC_v \cup SC_a} P(C) \land P(C^v_N) = \max_{C \in SC_v \cup SC_a \setminus C^a_M} P(C) \quad (7.5)
\]

In the input stream the packets storing a video frame are interleaved with the packets storing \( N_a \) audio frames. Because of this interleaving, the processing of the packets storing a video frame in the video sub-chain is interleaved with the processing of the audio packets in the audio sub-chain as well. The processing
of a video frame in the video sub-chain must be completed within $2 \times T_N$ so that the system outputs each video frame at the correct rate. The relevant question is how many audio frames $N_a$ can be processed interleaved with the video frame so the additional load on the CPU due to this interfering processing does not make the video frame miss its deadline.

We assume that the interference of the audio sub-chain during the processing of a video frame to the end of the video sub-chain does not cause missing the deadline imposed by the video renderer. This is expressed as:

$$S_{pv}^M + N_a \times T_M < 2 \times T_N$$

(7.6)

$S_{pv}^M$ represents the maximum processing time between the production of two consecutive packets in $f_{kn}$ not including the interfering, interleaved processing of the audio sub-chain. The above relation implies that

$$N_a < \left( \frac{2 \times T_N - S_{pv}^M}{T_M} \right) < \frac{2 \times T_N}{T_M} < 2.$$  

(7.7)

This implies that during the processing of 1 video frame, less than 2 audio frames can be processed such that the execution of the video sub-chain does not miss deadlines. The condition can be satisfied by constructing the stream such that packets storing one video frame are interleaved with packets storing always less than 2 audio frames (actually less than $2 \times T_N/T_M$ audio frames). $N_a \geq 1$ because during each $T_M$ 1 audio frame must be processed to the end of the audio sub-chain. $T_M < 2 \times T_N$ hence within each $2 \times T_N$ 1 audio frame must be processed to the end of the audio sub-chain.

Next we express $N_a$ in terms of the packets storing the audio frames interleaved with the packets storing the video frame. In a worst case situation of interleaving in which each video packet would be followed by a sequence of $X_a$ audio packets, the equation above is expressed as

$$S_{pv}^M + (F_v \times X_a/f_a) \times T_M < 2 \times T_N$$

(7.8)

which implies that

$$F_v \times X_a/f_a < \left( \frac{2 \times T_N - S_{pv}^M}{T_M} \right) < \frac{2 \times T_N}{T_M} < 2.$$  

(7.9)

where $F_v \times X_a/f_a$ represents the maximum number of audio frames interleaved with the packets storing the video frame. Note that when $F_v > 1$ follows that

$$X_a/f_a < \frac{2 \times T_N}{(T_M \times F_v)} < 1.$$  

(7.10)

which is to be interpreted that $X_a$ packets always make up less than 1 one encoded audio frame.

Conversely, in the case of the audio sub-chain execution, we assume that the interference of the video sub-chain during the processing of an audio frame
7.3 QoS requirements

The QoS requirement of this system is satisfied when the QoS requirements of both sub-chains $SC_v$ and $SC_a$ are satisfied. Hence the QoS requirement for to the end of the video sub-chain does not cause missing the deadline imposed by the audio renderer. This is expressed as:

$$S^M_{pa} + N_v \times 2 \times T_N < T_M \quad (7.11)$$

where $N_v$ represents the number of video frames processed to the end of the video sub-chain within $T_M$. This execution is interleaved with the processing of the audio frame to the end of its sub-chain. $S^M_{pa}$ represents the maximum processing time between the production of two consecutive packets in $f_{qa}^M$ not including the interfering, interleaved processing of the video sub-chain. The above relation implies that

$$N_v < \frac{(T_M - S^M_{pa})}{2 \times T_N} < \frac{T_M}{2 \times T_N} < 1. \quad (7.12)$$

This implies that during the processing of 1 audio frame less than 1 video frame can be processed such that the execution of the audio sub-chain does not miss deadlines. The condition can be satisfied by constructing the stream such that packets storing one audio frame are interleaved with packets storing always less than 1 video frame.

When expressing $N_v$ in terms of the packets storing the video frame interleaved with the packets storing the audio frame, in a worst case situation where each audio packet would be followed by a sequence of video packets, the equation above is expressed as

$$S^M_{pa} + (F_a \times X_v/f_v) \times 2 \times T_N < T_M \quad (7.13)$$

which implies that

$$F_a \times X_v/f_v < \frac{(T_M - S^M_{pa})}{2 \times T_N} < \frac{T_M}{2 \times T_N} < 1. \quad (7.14)$$

where $F_a \times X_v/f_v$ represents the maximum number of video frames interleaved with the packets storing the audio frame. Note that

$$X_v/f_v < 1. \quad (7.15)$$

which is to be interpreted that $X_v$ packets always make up less than 1 encoded video frame. Also we choose that the size of packets in $f_{q}^v$ and $f_{q}^a$ is smaller than the minimum size of an encoded frame in $SC_v$ and $SC_a$. Equation (7.16) expresses that the sum of $S^M_{pv}$ and $S^M_{pa}$ the processing time is smaller than $T_M$.

$$S^M_{pv} + S^M_{pa} < T_M \quad (7.16)$$

7.3 QoS requirements

The QoS requirement of this system is satisfied when the QoS requirements of both sub-chains $SC_v$ and $SC_a$ are satisfied. Hence the QoS requirement for
the entire system is defined as
\[ SQoSc_{\text{system}}(t, \text{pref}) \equiv SQoSc(t, \text{pref}, f_{q_{N-1}}^v, f_{q_{N-1}}^a, b_{q_{N}}^v, f_{q_{N}}^v, T_0^N, T_N) \land \]
\[ SQoSc(t, \text{pref}, f_{q_{M-1}}^a, f_{q_{M}}^a, T_0^M, T_M). \]
where \( \text{pref} \sim t \in \text{St}(\rho) \). We approach for now only QoS aspects as have been specified in Chapter 4. At the end of the current chapter we address issues related to synchronization between the two sub-chains.

### 7.4 System execution analysis

We analyze the execution of each of the two sub-chains \( SC_v \) and \( SC_a \).

**Lemma 7.1.** Consider the \( SC_v \) sub-chain. When \( S_v^M < 2T_N \) then there exists \( t_{\text{init}}^v \in \text{St}(\rho) \) such that \( C_i b_{q_i} \) (in \( t_{\text{init}}^v \) of \( T_{cc} \), \( \forall C_i \in SC_v, i < N \)).

**Proof.**

From the perspective of sub-chain \( SC_v \), the effect of the demultiplexer component is to take as input a sequence of \( x + 1 \) packets ending with a video packet, and to replace this sequence with this last (video) packet. In a worst case situation from the perspective of \( SC_v \), the number of packets preceding the video packet is equal to \( X_v \).

The worst case computation time necessary for producing the video packet in queue \( f_{q_v}^v \) must take into account the worst case time for processing the \( X_v \) packets to the end of the \( SC_a \) sub-chain, which in a worst case situation may precede (depending on the priority assignment) the production of the video packet. The worst case time for processing the \( X_v \) packets to the end of \( SC_a \) is
\[ X_v / f_a \ast T_M \]

Hence the worst case computation time necessary for producing the last packet in queue \( f_{q_v}^v \) is
\[ S_v^M = S_v^M (t_{C_2}^v) + X_v / f_a \ast T_M \]  
(7.17)

In fact from the perspective of the \( SC_v \) sub-chain, (7.17) gives the worst case computation time of \( C_2 \) used for processing a packet in \( f_{q_2}^v \). When
\[ S_v^M < 2T_N \]
where \( S_v^M \) includes the value of \( S_v^M \) which is the the worst case computation time of \( C_2 \) used for processing a packet in \( f_{q_2}^v \).

\( S_v^M < 2T_N \) implies close similarities between the behaviour of sub-chain \( SC_v \) and the system described in Chapter 6. In each of these situations there are inserted more packets than needed to decode 1 frame. For this reason we can use the same approach as used for Lemma 6.2 to show the filling up of the forward queues in the system. \( \Box \)
Next we wish to specify the executions of sub-chains $SC_v$ and $SC_a$ within the entire system. We denote the traces recording the individual executions of sub-chains $SC_v$ and $SC_a$ with $\rho_v$ and respectively $\rho_a$. Traces $\rho_v$ and $\rho_a$ are projections of $\rho$ on the union of alphabets of components in $SC_v$ and respectively $SC_a$:

$$\rho_v = \rho \uparrow \bigcup_{C \in SC_v} A'(C)$$

$$\rho_a = \rho \uparrow \bigcup_{C \in SC_a} A'(C)$$

The following theorem states that $\rho_v$ has a repetitive nature according to the $x:1$ parameterized pattern studied in Chapter 6, and presents in detail the composition of each iteration of this repetitive trace.

**Theorem 7.1.** Consider the $SC_v$ chain illustrated in Figure 7.1. Given that $S_v^M < 2T_N$ holds, after a finite prefix $t_{\text{init}}^v$, the trace $\rho_v$ recording the execution of the chain will adopt a parameterized pattern (with parameter $x$) of chain execution corresponding to the two behaviours of $C^v_3$ according to its $x:1$ input-output relation. Depending on the nature of the input stream $\rho$, is:

$$\rho_v = t_{\text{init}}^v \dashv t_{\text{stable}}^v$$

where

$$t_{\text{stable}}^v = (s_{x:1})^g$$

Sub-trace $s_{x:1}$ specifies the execution of the chain processing portions of the stream that determine an $x:1$ input-output relation for $C^v_3$:

$$s_{x:1} = t_{C_v}^1 \dashv t_{LSCN}^l \dashv d(T_0^N + i \ast T_N) \dashv t_{C_v}^2 \dashv t_{LSCN}^l \dashv d(T_0^N + i \ast T_N)$$

- $t_{LSCN}^l$ and $t_{LSCN}^r$ are sub-traces of $t_{LSCN}$ and $t_{LSCN}^l \dashv t_{LSCN}^r = t_{LSCN}$ where

- $t_{LSCN}^l$ records the interleaved execution of components $C^v_3, \ldots, C^v_{N-1}$.

- $t_{C_v}^1 = \exists \text{ end frame} = \text{NULL \#} X$.

- $t_{C_v}^{1,1}' = \exists' \text{ end frame} = \text{NULL \#} X$.

- $t_{C_v}^{1,1}C_3' \text{ records the interleaved execution of actions in:}$

  - $(t_{C_v}^{1,1})^r \dashv rf(p, \text{ frame})^1 \dashv \ldots \dashv rf(p, \text{ frame})^{m_2}$.

  - $\exists' fq_j \dashv \text{ true} \dashv fq_{j-1} $ where $P(C^v_j) < P(C^v_1)$, $C^v_j \in SC_v$, $3 < j < N$.

  - $(t_{C_v}^2)^x$.

- $(t_{C_v})^x$.

- $t_{C_v}^{1,1}C_3' \text{ records the interleaved execution of actions in:}$

- The exact order of actions in $t_{C_v}^{1,1}C_3'C_2'C_1$ is determined by the priorities of $C_3 C_2, C^v_j$ and $C_1$, the computation times of actions of components in $SC_v$, and the length of the deferral times of $C_1$. 


• At the end of $s_{x:1}$ all components $C_i$ are blocked at $bq_i^y$? ($C_1 \ b q_i^y$), $\forall C_i \in SC_v$, $i < N$.

Proof. $C_3^v$ can execute only according to the $x:1$ pattern because the size of packets in $fq_v^y$ is smaller than the minimum size of an encoded frame in $SC_v$.

Similar approach as used for Theorem 6.2.

Lemma 7.2. Consider the $SC_a$ sub-chain. When $S_M^a < T_M$ then there exists $t_{init}^a \in St(p)$ such that $C_i \ b q_i^y$? ($C_1 \ b q_i^y$) [in $t_{init}^a$ of $T_cc$], $\forall C_i \in SC_a$, $i < M$.

Proof. Identical approach as for Lemma 7.1.

Theorem 7.2 specifies the individual execution of $SC_a$ within the overall system by presenting the nature and composition of trace $\rho_a$. Although the insights given by Theorem 7.1 and Theorem 7.2 are similar, and traces $\rho_v$ and $\rho_a$ are resembling, they are not identical because the time-driven component in $SC_v$ executes according to the interlaced standard, while the time-driven component in $SC_a$ does not. For the purpose of thoroughness we present and characterize both traces $\rho_v$ and $\rho_a$ in the two theorems.

Theorem 7.2. Consider the $SC_a$ chain illustrated in Figure 7.1. Given that $S_M^a < T_M$ holds, after a finite prefix $t_{init}^a$, the trace $\rho_a$ recording the execution of the chain will adopt a parameterized pattern (with parameter $x$) of chain execution corresponding to the two behaviours of $C_3^a$ according to its $x:1$ input-output relation. Depending on the nature of the input stream $\rho_a$ can be:

$$\rho_a = t_{init}^a \sim t_{stable}^a$$

where

$$t_{stable}^a = (s_{x:1})^a$$

Sub-trace $s_{x:1}$ specifies the execution of the chain processing portions of the stream that determine an $x:1$ input-output relation for $C_3^a$:

$$s_{x:1} = t_{CM}^a \sim t_{LSCM}^a \sim d(T_M + i \ast T_M)$$

where

$$t_{LSCM}^a = t_L^a \sim t_{md}^a \sim t_{1:1}^a \sim t_{x:1}^a \sim t_{C1C2C3}^ad$$

and

• $t_L^a$ records the interleaved execution of components $C_3^a, \ldots, C_{M-1}^a$.
• $t_{C3}^a = \chi$ end_frame = NULL $\chi$.
• $t_{x:1}^a \sim t_{C1C2C3}^a$ records the interleaved execution of actions in:
7.4 System execution analysis

- \( (t_C^1)^{tx} \rightarrow rf(p, \text{frame})^1 \leftarrow \ldots \rightarrow rf(p, \text{frame})^m_{N2} \).
- \( \exists f_j \vdash true \leftarrow f_j,_{j-1} \vdash \) where \( P(C_j^v) < P(C_1) \), \( C_j^v \in SC_v \), \( 3 < j < N \).
- \( (tC_2)^x \).
- \( (tC_1)^x \).

- The exact order of actions in \( tx^1 \) is determined by the priorities of \( C_2, C_0 \) and \( C_1 \), the computation times of actions of components in \( SC_o \), and the length of the deferral times of \( C_1 \).

- At the end of \( sx^1 \) all components \( C_i \) are blocked at \( bq^v \) \( (C_1 \ b \ bq^1 ?) \), \( \forall C_i \in SC_a, i < M \).

Proof. \( C_2^v \) can execute only according to the \( x^1 \) pattern because the size of packets in \( f_q^2 \) is smaller than the minimum size of an encoded frame in \( SC_o \). Similar approach as used for Theorem 6.2.

We consider now \( t_{\text{init}} \) the shortest prefix of \( p \) in which both sub-chains have reached their stable phase. The following theorem describes the execution of the combined system. It describes all possibilities of interleaved execution of the two \( x^1 \) stable phases of sub-chain \( SC_v \) and \( SC_o \) within any interval \([T_0^N + 2lT_N, T_0^N + 2(l + 1)T_N], l \in \mathbb{N} \). The actual interleaving is determined by the ratio between \( T_M \) and \( 2T_N \), the contents of the input stream which influences the computation times of the trace actions, the duration of the deferral times of \( C_1 \) and the priority assignment of the components. For each of the eight cases describing possibilities of interleaving we present the actual trace and we study whether the overall system \( QoS \) requirement is satisfied. The theorem shows that among the eight cases we distinguish, in three of them \( (where C_M^p \) preempts \( C_N^v \) ) the interleaving of components has no negative influence on the measure in which the \( QoS \) requirement is satisfied, while in five cases the \( QoS \) requirement for \( SC_v \) is satisfied only under the condition that:

\[
S_M^M(t_{C_N^v}^1) + S_M^M(t_{C_N^v}) < flush.
\]

The interleaving cases have been obtained by progressively shifting in time the two pattern of execution described in the previous two theorems which describe the individual behaviour of each two sub-chains. We present below the trace interleaving in the more complex case where \( S_M^M > T_N \). The \( S_M^M \leq T_N \) simpler case is to be solved analogously.

**Theorem 7.3.** Consider a system as in Figure 7.1. Under the assumptions presented in section 7.2, within any interval \([T_0^N + 2lT_N, T_0^N + 2(l + 1)T_N], l \in \mathbb{N} \),
the system trace \( \rho \) records the interleaving of traces \( \rho_v \) and \( \rho_a \) in the following way:

\[
\rho = t_{\text{init}} \prec t_{\text{spi}}
\]

with \( t_{\text{spi}} \) recording the stable phases interleaving \( \rho^u_{\text{stable}} \) and \( \rho^a_{\text{stable}} \) of \( SC_v \) and \( SC_a \). Trace \( t_{\text{spi}} \) is expressed as

\[
t_{\text{spi}} = u \prec t \prec v
\]

where \( t \) records a possible interleaving of \( \rho^u_{\text{stable}} \) and \( \rho^a_{\text{stable}} \) within an interval \([T_0^N + 2lT_N, T_0^N + 2(l+1)T_N], \) \( l \in \mathbb{N} \).

**A.** When \( t_{CN}^k \) starts executing during the odd periods of each \( 2T_N \):

\[
T_0^N + 2lT_N < \sigma(t_{CN}^k) - S_8(t_{CN}^k) < T_0^N + (2l+1)T_N, \ k, l \in \mathbb{N} \quad (7.18)
\]

1. \( t_{CN}^k \) and \( t_{CN}^{k+1} \) are executed within interval \([T_0^N + 2lT_N, T_0^N + 2(l+1)T_N]):

\[
\sigma(t_{CN}^k) - S_8(t_{CN}^k) + T_M + S_\delta(t_{CN}^{k+1}) \leq T_0^N + 2(l+1)T_N \quad (7.19)
\]

**a.** The execution of \( t_{CN}^k \) starts after end execution of \( t_{CN}^{1, i} \):

\[
\sigma(t_{CN}^k) - S_\delta(t_{CN}^k) \geq (T_0^N + 2lT_N) + S_\delta(t_{CN}^{1, i}) \quad (7.20)
\]

then

- \( t = t_{CN}^{1, i} \prec t_{ab}^1 \prec d(T_0^M + kT_M) \prec t_{CN}^2 \prec t_{ab}^2 \prec d(T_0^N + (2l+1)T_N) - (t_{CN}^{2})^l \prec t_{ab}^3 \prec d(T_0^N + (k+1)T_M) \prec t_{CN}^{k+1} \prec t_{LSCN}^{k+1} \prec d(T_0^N + 2(l+1)T_N). \)

- \( SQoSc_{\text{system}}(t, \ \text{pref}) \) holds, where \( \text{pref} \prec t \in St(\rho) \) and \( t_{\text{init}} \subseteq \text{pref} \).

where \( t_{ab}^1 \prec t_{ab}^2 \prec t_{ab}^3 = t_{ab} \) and \( t_{ab} \) records the interleaved execution of the following traces: \( (LSCN)_l \), \( t_{LSCN}^2 \), \( t_{LSCN}^3 \).

**b.** The execution of \( t_{CN}^k \) starts during execution of \( t_{CN}^{1, i} \):

\[
T_0^N + 2lT_N < \sigma(t_{CN}^k) - S_8(t_{CN}^k) < (T_0^N + 2lT_N) + S_8(t_{CN}^{1, i}) \quad (7.21)
\]
7.4 System execution analysis

\[ t = (t_{C_N}^{1'}) \sim d(T_0^M + k \cdot T_M) \sim t_{CM}^k \sim (t_{C_N}^{1'})'' \sim t_{ab}^1 \sim d(T_0^N + (2l + 1)T_N) \sim (t_{C_N}^{2'}) \sim t_{ab}^2 \sim d(T_0^M + (k + 1)T_M) \sim t_{CM}^{k+1} \sim t_{LSCM}^{k+1} \sim d(T_0^N + 2(l + 1)T_N). \]

- \( SQoS(t, \text{pref}, f^{M-1}_q, f^{M}_q, T_0^M, T_M) \) holds.
  If \( S^M_{\delta}(t_{C_N}^{1'}) + S^M_{\delta}(t_{CM}^k) < \text{flush} \) then
  \( SQoS(t, \text{pref}, f^{M-1}_q, f^{M}_q, t_{CM}^k, T_0^N, T_N) \) holds as well, thus
  \( SQoS_{\text{system}}(t, \text{pref}) \) holds, where \( \text{pref} \sim t \in \text{St}(p) \) and \( t_{\text{init}} \subseteq \text{pref} \).

where \( t_{C_N}^{1'} = (t_{C_N}^{1'})' \sim (t_{C_N}^{1'})'' \) and \( t_{ab}^1 \sim t_{ab}^2 = t_{ab} \). Trace \( t_{ab} \) records the interleaved execution of the following traces: \( (t_{LSCN}^{1'})', (t_{LSCN}^{k})', (t_{LSCN}^{2'})' \).

\[ \sigma(t_{CM}^k) - S_{\delta}(t_{CM}^k) = T_0^N + 2lT_N \quad (7.22) \]

\[ t = t_{CM}^k \sim d(T_0^M + 2lT_N) \sim t_{C_N}^{1'} \sim t_{ab}^1 \sim d(T_0^N + (2l + 1)T_N) \sim (t_{C_N}^{2'}) \sim t_{ab}^2 \sim d(T_0^M + (k + 1)T_M) \sim t_{CM}^{k+1} \sim t_{LSCM}^{k+1} \sim d(T_0^N + 2(l + 1)T_N). \]

- \( SQoS(t, \text{pref}, f^{M-1}_q, f^{M}_q, T_0^M, T_M) \) holds.
  If \( S^M_{\delta}(t_{C_N}^{1'}) + S^M_{\delta}(t_{CM}^k) < \text{flush} \) then
  \( SQoS(t, \text{pref}, f^{M-1}_q, f^{M}_q, t_{CM}^k, T_0^N, T_N) \) holds as well, thus
  \( SQoS_{\text{system}}(t, \text{pref}) \) holds, where \( \text{pref} \sim t \in \text{St}(p) \) and \( t_{\text{init}} \subseteq \text{pref} \).

where \( t_{C_N}^{1'} = (t_{C_N}^{1'})' \sim (t_{C_N}^{1'})'' \) and \( t_{ab}^1 \sim t_{ab}^2 = t_{ab} \). Trace \( t_{ab} \) records the interleaved execution of the following traces: \( (t_{LSCN}^{1'})', (t_{LSCN}^{k})', (t_{LSCN}^{2'})' \).

2. \( t_{CM}^k \) and \( t_{CM}^{k+1} \) are not executed within the same interval.
Figure 7.5. Case A. 1. c.

\[ T_0^N + 2IL_N, T_0^N + 2(l + 1)T_N: \]
\[ \sigma(t_{CM}^k) - S_B(t_{CM}^k) + T_M > T_0^N + 2(l + 1)T_N \]  \hspace{1cm} (7.23)

\textbf{a. The execution of } t_{CM}^k \text{ starts after end execution of } t_1^{1, l}: \]
\[ \sigma(t_{CM}^k) - S_B(t_{CM}^k) \geq (T_0^N + 2IT_N) + S_B(t_1^{1, l}) \]  \hspace{1cm} (7.24)

\textbullet \quad \text{then}
- \quad \text{SQoS}_{\text{system}}(t, \text{pref}) \text{ holds, where } \text{pref} \in \text{St}(p) \text{ and } t_{init} \subseteq \text{pref.}

\text{where } t_1^{1, l} - t_2^{1, l} - t_3^{1, l} = t_{ab} \text{ and } t_{ab} \text{ records the interleaved execution of the}
\text{following traces: } (t_{LSCM}^{l})^l, t_{LSCM}^k, (t_{LSCM}^{l'})^l.

Figure 7.6. Case A. 2. a.

\textbf{b. The execution of } t_{CM}^k \text{ starts during execution of } t_1^{1, l}: \]
\[ T_0^N + 2IT_N < \sigma(t_{CM}^k) - S_B(t_{CM}^k) < (T_0^N + 2IT_N) + S_B(t_1^{1, l}) \]  \hspace{1cm} (7.25)

\text{then}
- \quad t = (t_{CM}^{1, l})^l \sim d(T_0^M + k * T_M) \sim t_{CM}^k \sim (t_{CM}^{1, l})^n \sim
7.4 System execution analysis

\[
t_{ab}^1 - d(T_0^N + (2l + 1)T_N) - (t_{CN}^2) - t_{ab}^2 - d(T_0^N + 2(l + 1)T_N).
\]

- \(SQoSc(t, \text{pref}, f_{q_{M-1}}, f_{q_{M}}, T_0^N, T_M)\) holds.

If \(S_d^M(t_{CN}^1) + S_d^M(t_{CM}^1) < \text{flush}\) then

\(SQoSci(t, \text{pref}, f_{q_{N-1}}, f_{q_{N}}, b_{q_{N}}, f_{q_{N}^1}, T_0^N, T_N)\) holds as well, thus

\(SQoSc_{system}(t, \text{pref})\) holds, where \(\text{pref} - t \in St(\rho)\) and \(t_{init} \subseteq \text{pref}\).

where \(t_{a}^1 = (t_{CN}^2) - (t_{CN}^2)'\) and \(t_{a}^2 - t_{ab}^2 = t_{ab}\). Trace \(t_{ab}\) records the interleaved execution of the following traces: \((t_{LSCa})^1, (t_{LSCa}^2), (t_{LSCa}^2)'\).

Figure 7.7. Case A. 2. b.

\textbf{B.-} \(t_{CM}^k\) starts executing during the even periods of each \(2T_N\):

\[
T_0^N + (2l + 1)T_N \leq \sigma(t_{CM}^k) - S_d(t_{CM}^k) < T_0^N + 2(l + 1)T_N \quad (7.26)
\]

1. \(t_{CN}^k\) and \(t_{CN}^{k-1}\) are executed within interval \([T_0^N + 2lT_N, T_0^N + 2(l + 1)T_N]\):

\[
\sigma(t_{CM}^k) - S_d(t_{CM}^k) - T_M \geq T_0^N + 2lT_N \quad (7.27)
\]

This case is equivalent with case A.1. and the execution and execution properties of the system are the same as mentioned at A.1.a., A.1.b., A.1.c.

2. \(t_{CM}^k\) and \(t_{CM}^{k-1}\) are not executed within interval

\([T_0^N + 2lT_N, T_0^N + 2(l + 1)T_N]\):

\[
\sigma(t_{CM}^k) - S_d(t_{CM}^k) - T_M < T_0^N + 2lT_N \quad (7.28)
\]

\textbf{a.} The execution of \(t_{CM}^k\) starts after end execution of \(t_{CN}^2\):

\[
\sigma(t_{CM}^k) - S_d(t_{CM}^k) \geq (T_0^N + (2l + 1)T_N) + S_d(t_{CN}^2) \quad (7.29)
\]

then

- \(t = t_{CN}^1 - t_{ab}^1 - d(T_0^N + (2l + 1)T_N) - (t_{CN}^2) - t_{ab}^2 - d(T_0^M + kT_M) - t_{CM}^k - t_{LSCM}^k - d(T_0^N + 2(l + 1)T_N).\)
• $SQoSc_{system}(t, \text{pref})$ holds, where $\text{pref} \sim t \in St(\rho)$ and $t_{init} \subseteq \text{pref}$. Where $t_{ab}^1 \prec t_{ab}^2 = t_{ab}$ and $t_{ab}$ records the interleaved execution of the following traces: $t_{LSC_M}^k, (t_{LSC_M}^l')^l, (t_{LSC_M}^{l''})^l$.

\[ T_0^N + (2l + 1)T_N < \sigma(t_{LSC_M}^k) - S_8(t_{LSC_M}^k) < (T_0^N + (2l + 1)T_N) + S_8(t_{LSC_M}^{l''}) \quad (7.30) \]

then

• $t = t_{C_N}^{l'} \prec t_{ab}^1 \prec d(T_0^N + (2l + 1)T_N)$

\[ (t_{C_N}^{l'})' \prec d(T_0^N + k \times T_M) \prec t_{C_M}^k \prec (t_{C_N}^{l''})'' \prec t_{ab}^2 \prec d(T_0^N + 2(l + 1)T_N). \]

• $SQoSc(t, \text{pref}, f_{dM-1}^q, f_{dM}^q, f_{qM}^q, f_{qM}^q, T_0^N, T_M)$ holds.

If $S_8^{M-1}(t_{C_N}^{l'}) + S_8^{M}(t_{C_N}^{l''}) < \text{flush}$ then

$SQoSc(t, \text{pref}, f_{qN-1}^q, f_{qN}^q, f_{qN}^q, f_{qN}^q, T_0^N, T_M)$ holds as well, thus $SQoSc_{system}(t, \text{pref})$ holds, where $\text{pref} \sim t \in St(\rho)$ and $t_{init} \subseteq \text{pref}$. Where $t_{ab}^2 = (t_{C_N}^{l''})' - (t_{C_N}^{l''})' \prec (t_{ab}^2)^l$ and $t_{ab}^1 \prec t_{ab}^2 = t_{ab}$. Trace $t_{ab}$ records the interleaved execution of the following traces: $t_{LSC_M}^k, (t_{LSC_M}^l')^l, (t_{LSC_M}^{l''})^l$.
7.4 System execution analysis

c. The execution of \( t_{CM}^k \) starts at moment \( T_0^N + (2l+1)T_N \):

\[
\sigma(t_{CM}^k) - S_6(t_{CM}^k) = T_0^N + (2l+1)T_N
\]  

(7.31)

then

- \( t = t_{CN}^1 - t_{ab} - d(T_0^M + k \times T_M) \)

\[
t_{CN}^2 - d(T_0^N + (2l+1)T_N) - t_{CN}^2 - t_{ab} - d(T_0^N + 2(l+1)T_N).
\]

- \( \text{SQoSc}(t, \text{pref}, f_{qM}^{d-1}, f_{qM}^M, T_0^M, T_M) \) holds.

If \( S(\sigma(t_{CN}^1)) + S(\sigma(t_{CN}^2)) < \text{flush} \) then

\( \text{SQoSc}(t, \text{pref}, f_{qN}^{d-1}, f_{qN}^N, b_{qN}^N, f_{qN}^N, T_0^N, T_N) \) holds as well, thus

\( \text{SQoSc}_{\text{system}}(t, \text{pref}) \) holds, where \( \text{pref} = t \in \mathcal{St}(\rho) \) and \( t_{\text{init}} \subseteq \text{pref} \).

where \( t_{CN}^2 = (t_{CN}^2)^l \) and \( t_{ab} - t_{ab} = t_{ab} \). Trace \( t_{ab} \) records the interleaved execution of the following traces: \( (t_{LSCN})^l, t_{LSCM}^k, (t_{LSCN})^N \).

Figure 7.10. Case B. 2. c.

**Proof.** We prove the theorem statement by construction of the trace \( t \) describing the execution of the system during the time interval \( [T_0^N + 2lT_N, T_0^N + 2(l+1)T_N] \).

**A.1.a.** Given equation (7.20) follows that at moment \( T_0^N + 2lT_N \) the highest priority, ready-to-run component in the system is \( C_N \). \( C_N^v \) executes \( t_{CN}^1 \) according to its program. During \( t_{CN}^1 \), \( C_N^v \) executes \( b_{qN}^N \) which triggers a de-blocking in cascade of components in \( LSC_N \). Their execution is recorded until moment \( T_0^N + (2l+1)T_N \) by sub-trace \( (t_{LSCN})^l \).

Again given equations (7.20) and (7.5) follows that the execution of \( (t_{LSCN})^l \) is preempted by \( d(T_0^M + k \times T_M) \) at moment \( T_0^M + k \times T_M \). The partial execution of \( (t_{LSCN})^l \) up until the preemption is denoted by \( t_{ab}^l \). The delay action is followed by \( t_{CM}^k \) of \( C_M \). The execution of \( t_{CM}^k \) ends before moment
$T_0^N + (2l+1)T_N$ because of the condition expressed in (7.18). During $t_{C_M}^l$ component $C_M$ executes $bq_{M-l+1}$ which triggers a de-blocking in cascade of components in $LSC_M^l$. Their execution is recorded by $t_{LSC_M}^l$. After $t_{C_M}^l$ traces $(t_{LSC_M}^l)^l$ and $t_{LSC_M}^l$ execute interleaved and their interleaved execution until moment $T_0^N + (2l+1)T_N$ is denoted by $t_{ab}^l$.

At this point we can express $t$ as:

$$t = t_{C_N}^l t_{ab}^l - d(T_0^M + k * T_M) - t_{C_M}^l - t_{ab}^l - w.$$  

At moment $T_0^N + (2l+1)T_N$, given (7.5), the second delay action of $C_N$, $d(T_0^N + (2l+1)T_N)$ is executed, followed by $(t_{C_N}^l)^l$. The execution of $(t_{C_N}^l)^l$ is followed by $(t_{LSC_N}^l)^l$ according to the $\rho$ trace specified in Theorem 7.1. Trace $(t_{LSC_N}^l)^l$ may still be interleaved with $t_{LSC_M}^l$, depending on the priority assignment of all components in the system. This interleaved behaviour is denoted by $t_{ab}^l$. So far $t$ can be expressed as:

$$t = t_{C_N}^l t_{ab}^l - d(T_0^M + k * T_M) - t_{C_M}^l - t_{ab}^l - d(T_0^N + (2l+1)T_N) - (t_{C_N}^l)^l - t_{ab}^l - w.$$  

The execution of $t_{ab}^l$ cannot exceed moment $T_0^M + (k+1)(T_M)$ because of the condition expressed in (7.16). The end of $t_{ab}^l$ also marks the end of $(t_{LSC_N}^l)^l$ and $t_{LSC_M}^l$ when according to Theorem 7.1 and Theorem 7.2 all components have become blocked at receiving an empty packet.

At moment $T_0^M + (k+1)T_M$, given that all components are blocked the delay action of $C_M$ is executed followed by $t_{C_M}^{k+1}$. We know that the execution of $t_{C_M}^{k+1}$ precedes moment $T_0^N + 2(l+1)T_N$ because of (7.19). Finally, at moment $T_0^N + 2(l+1)T_N$, given (7.5), the delay action of $C_N$, $d(T_0^N + 2(l+1)T_N)$ is executed. Therefore $t$ can be expressed as:

$$t = t_{C_N}^l t_{ab}^l - d(T_0^M + k * T_M) - t_{C_M}^l - t_{ab}^l - d(T_0^N + (2l+1)T_N) - (t_{C_N}^l)^l - t_{ab}^l - d(T_0^M + (k+1)T_M) - t_{C_M}^{k+1} - t_{LSC_M}^{k+1} - d(T_0^N + 2(l+1)T_N).$$  

Given the reasoning above illustrated in Figure 7.3, we also conclude that $SQsoS_{system}(t, preff)$ holds. Cases A.1.b. and A.1.c. are to be proved in the same manner as above.

A.2.a. The execution specified by $t$ in this case is identical up to the end of trace $t_{ab}^l$ with the one specified at case A.1.a. for the same reasons as presented in that case. Also in this case at the end of $t_{ab}^l$ all components have become blocked at receiving an empty packet. However at this point given (7.23) fol-
7.4 System execution analysis

Lows that the execution of
\[ d(T_0^M + (k + 1)T_M) \sim t_{C_M}^{k+1} \sim t_{LSC_M}^{k+1} \]
is postponed for the next \( 2T_N \) interval and \( t_{ab}^1 \) is followed only by \( d(T_0^N + 2(l + 1)T_N) \).

Therefore in this case \( t \) can be expressed as:
\[
\begin{align*}
    t & = t_{C_N}^1 - t_{ab}^1 - d(T_0^M + k * T_M) - t_{C_N}^2 - t_{ab}^2 \\
    & - d(T_0^N + (2l + 1)T_N) - (t_{C_N}^2)^l - t_{ab}^2 \\
    & - d(T_0^N + 2(l + 1)T_N).
\end{align*}
\]

Given the reasoning above illustrated in Figure 7.6, we conclude that \( SQoS_{system}(t, pre) \) holds. Cases A.2.b. is to be proved in the same manner as above.

B.1.a. The equivalence between this case and case A.1. results directly given the equivalence of the equations characterizing the two cases.

B.2.a. Given equation (7.29) follows that at moment \( T_0^N + 2T_N \) the highest priority, ready-to-run component in the system is \( C_N \). \( C_N \) executes \( t_{ab}^1 \) according to its program. During \( t_{C_N}^1 \), \( C_N \) executes \( bq_{N-1} \) which triggers a de-blocking in cascade of components in \( LSC_N \). Their execution is recorded until moment \( T_0^N + (2l + 1)T_N \) by sub-trace \( (t_{LSC_N}^1)^l \). Depending on the computation times of actions in \( t_{LSC_N}^1 \), trace \( (t_{LSC_N}^1)^l \) may execute interleaved with part of \( t_{LSC_N}^{k-1} \) which starts in the previous \( 2T_N \) interval as shown in all A.1. cases. This interleaved behaviour is denoted by \( t_{ab}^1 \).

Given (7.5) follows that the execution of \( t_{ab}^1 \) is followed at moment \( T_0^N + (2l + 1)T_N \) by \( d(T_0^N + (2l + 1)T_N) \) and \( (t_{C_N}^2)^l \). The execution of \( (t_{C_N}^2)^l \) is followed by \( (t_{LSC_N}^2)^l \) as specified in Theorem 7.1. Trace \( (t_{LSC_N}^2)^l \) may still be interleaved with \( t_{LSC_N}^{k-1} \), depending on the priority assignment of all components in the system. This interleaved behaviour is denoted by \( t_{ab}^2 \). So far \( t \) can be expressed as:
\[
\begin{align*}
    t & = t_{C_N}^1 - t_{ab}^1 - \\
    & - d(T_0^N + (2l + 1)T_N) - (t_{C_N}^2)^l - t_{ab}^2. \\
\end{align*}
\]

Given the condition expressed in (7.16), the execution of \( t_{ab}^2 \) ends before moment \( T_0^M + kT_M \). Considering (7.29), (7.28) and (7.5) follows that at moment \( T_0^M + kT_M \), after the execution of
\[
(t_{C_N}^2)^l - t_{ab}^2
\]
the delay action of \( C_M \) \( d(T_0^M + (k + 1)T_M) \) executes followed by \( t_{C_M}^{k+1} \) and
Finally, at moment $T_0^N + 2(l + 1)T_N$, given (7.5), the delay action of $C_N^v$, $d(T_0^N + 2(l + 1)T_N)$ is executed. Therefore $t$ can be expressed as:

$$
t = t_{C_N^v}^{k+1} - t_{ab}^{l} - d(T_0^N + (2l + 1)T_N) - (t_{C_N^v}^{l}) - t_{ab}^{2} - d(T_0^M + kT_M) - t_{C_M^u}^{k} - t_{LSC_M^u}^{k+1} - d(T_0^N + 2(l + 1)T_N).$$

Given the reasoning above illustrated in Figure 7.8, we also conclude that $SQoS_{system}(t, pref)$ holds. Cases B.2.b. and B.2.c. are to be proven in the same manner as above. □

### 7.5 Practical applications

The practical application regarding optimization of memory, CPU utilization, response time and NCS are derived from the practical applications mentioned in sections (4.2 and 5.3.1). We briefly remind below the optimizations applicable for this case.

- the minimum necessary and sufficient capacity of each queue to avoid deadlock in the chain is 1. (Corollary 3.3).
- due to the same reasons explained in section 4.2.3 response time of the 2 sub-chains cannot be improved by assigning the minimum priority to $C_1$ as suggested in Chapter 3 because that does not change the influence of the time-driven components $C_N^v$ and $C_M^u$.
- as shown in Theorem 3.3 the response time of the two sub-chains is reduced by reducing the capacities of queues preceding $C_N^v$ and $C_M^u$.
- the number of context switches occurring due to the interleaved execution of non-time-driven components within the individual execution of each of the 2 sub-chains during their stable phase can be reduced by assigning priorities as shown in Theorem 3.2b-(ii).
- the number of context switches occurring due to interleaved execution of the data driven components in $SC_N^v$ with actions of $C_N^v$ (during the stable phase) can only be reduced by one context switch. This is the context switch due to preemption when $C_N^v$ has a higher priority than other components in the chain. This context switch can be avoided by assigning to $C_N^v$ the lowest priority in the chain. This comes at the cost of a lower $QoS$ when $S_M^v > T_N$. A second option is to impose $S_M^v < T_N$. The above statement holds also in the case of components of $SC_u$.
- optimization of CPU utilization by using the deferral times during the execution deferral is achieved by not assigning $C_i$ the minimum priority as shown in Property 5.2.
7.6 Summary

In this chapter we have studied the execution of a system in which components are connected according to the branching topology as shown in Figure 7.1. We have divided the system into two sub-chains ($SC_v$ and $SC_a$), each consisting of a component with deferred execution and a demultiplexer component, a component whose execution depends on the content of the input stream, a number of data-driven components and end with a time-driven component. We have shown that, given the assumption considered, after a finite prefix, each of the two sub-chains reaches its own stable phase in which all components become driven by the period execution of the time-driven component at the end of the chain.

Finally we have characterized the overall execution of the entire system in Theorem 7.3 by describing all possibilities of interleaved execution of the two $x:1$ stable phases of sub-chain $SC_v$ and $SC_a$ within any interval $[T_0^N + 2lTN, T_0^N + 2(l+1)TN], l \in \mathbb{N}$. We have explained that the actual interleaving is determined by the ratio between $TM$ and $2TN$, the contents of the input stream which influences the computation times of the trace actions, the duration of the deferral times of $C_1$ and the priority assignment of the components.

We concluded this chapter by summarizing the practical applications inherited from the previous chapters and applicable in this case.
In this chapter we analyze the behaviour of systems composed of independent linear chains. Section 8.1 presents composition of chains without timing constraints. Both chains are composed only of data-driven components. We show that composing these chains is not advisable because after a finite prefix one of the chains experiences starvation.

Section 8.2 tackles composition of chains with timing constraints. In practice the first chain corresponds to a video decoding chain and the second chain to a surveillance application that saves on the hard-disk the images captured by the first component. Due to the fact that the two chains compete for the processor resource the measure in which each of them meet their QoS requirements is potentially affected. The challenge in this case is to find solutions for designing the composition of the chains such that both chains satisfy their QoS requirements. We show that certain priority assignments imply supplementing the buffer capacities in the chains which is costly. We propose and detail a cheaper solution in which the buffers do not need to be larger than one position each. Our solution to satisfying the QoS requirement is to impose a specific priority assignment to the components and to control the phasing between the executions of the two systems. We also show how to design a system such that the condition for the phasing is satisfied.
8.1 Composition of chains consisting of only data-driven components

Consider a system consisting of two linear chains composed of only data-driven components as described in Chapter 3. The priorities assigned in the overall system to each component are unique. We denote the one chain with $CN_a$ and the other with $CN_b$. We denote the components in $CN_a$ with $C^a_i$ and those in $CN_b$ with $C^b_i$. The components with minimum priority in each chain are denoted by $C^a_m$ and respectively $C^b_m$. The unique traces corresponding to the chains are $\rho_a$ and $\rho_b$. The unique trace of the system is denoted by $\rho$. The union of the alphabets of all components in $CN_a$ and $CN_b$ is denoted by $A_a$ and respectively $A_b$. The following lemma states that the chain with a lower minimum in priority relative to the minimum in priority of the other chain, after a finite prefix will be starved.

**Lemma 8.1.** $P(C^a_m) > P(C^b_m) \Rightarrow \exists s, t : \rho = s \triangle t$ such that $\#(t, a) = 0$, $\forall a \in A_b$.

**Proof.** We know from Corollary 3.1 that each chain reaches the stable phase when the component with minimum priority executes for the first time.

$P(C^a_m) > P(C^b_m)$ implies that $CN_a$ will reach stable the phase before $CN_b$.

We denote the state of $\rho$ at which $C^a_m$ executes for the first time with $s$:

$$\rho = s \triangle f_q^{a_m-1} \triangle t.$$  

After $CN_a$ reaches the stable phase, all components in $CN_b$ have a higher priority than $C^b_m$, therefore $C^b_m$ will never be able to execute:

$$\forall i, P(C^a_i) > P(C^b_i) \Rightarrow \#(t, a) = 0, \forall a \in A(C^b_m).$$

This implies that $CN_b$ will never reach stable phase and will be starved after the finite prefix $s$ of $\rho$:

$$\#(t, a) = 0, \forall a \in A_b.$$  

The conclusion to this part of the study is that having chains composed of only data-driven components share one processor is not advisable because eventually one of the chains will be starved.

8.2 Composition of chains with timing constraints

This section addresses the composition of the two independent chains with timing constraints. One of the chains consists of a component with deferred execution, a component whose behaviour depends on the contents of the input, a number of simple data-driven components and a time-driven component. This type of chain has been studied in Chapter 6. The second chain that we
8.2 Composition of chains with timing constraints

Consider is composed of a time-driven component, a number of simple data-driven components and a component with deferred execution. This situation has been analyzed in section 5.4. The two chains execute on the same processor and compete for the system resources. In both cases the time-driven components execute according to the interlacing standard. In practice the first chain corresponds to a video decoding chain and the second chain to a surveillance application that saves on the hard-disk the images captured by the first component.

We denote again the set of component in the first chain with $C_{N_a}$ and the set of components in second chain with $C_{N_b}$. Then the set of all components in the system is $C_{N_a} \cup C_{N_b}$. The overall system execution is $\rho$. The traces recording the execution of the individual chains are $\rho_a$ and respectively $\rho_b$ where:

\[
\begin{align*}
\rho_a &= \rho \uparrow A_a \\
\rho_b &= \rho \uparrow A_b \\
\rho_a &= t_{\text{init}}^a \sim t_{\text{stable}}^a \\
\rho_b &= t_{\text{init}}^b \sim t_{\text{stable}}^b
\end{align*}
\]

where

\[
\begin{align*}
t_{\text{stable}}^a &= ((s_{1:x})^p \sim (s_{x+1})^k)^a \\
t_{\text{stable}}^b &= (t_b)^a
\end{align*}
\]

and

\[
t_r^b = \frac{1}{2} t_{C_1}^2 \sim t_{R_1} \sim d(T_0^1 + t_{C_1}^1) \sim t_{R_2} \sim d(T_0^1 + i * T_1)^k
\]

For $C_{N_a}$ we denote with $S_a^k$ the processing time between the production of two consecutive packets $k$ and $k-1$ in $f_{C_{N_a}}$ and with $S_a^M$ the maximum over all values $S_a^k$. Similarly $S_b^M$ denotes the same maximum sum for $C_{N_b}$. $T_1$ denotes the period of the $C_{N_a}^i$ time-driven component and $T_N$ is the period of $C_{N}^i$ As in Chapter 4, the period $T_N$ of the video renderer ($C_{N}^i$) is equal to the period $T_1$ of the video digitizer $C_{N_a}^i$:

\[
T_1 = T_N
\]

(8.1)

We also assume that

\[
S_a^M + S_b^M < 2T_1.
\]

(8.2)

Similar as in Chapter 4 the time-driven component that provides input has a higher priority than the one that produces output:

\[
P(C_{N_a}^i) = \max_{C \in C_{N_a} \cup C_{N_b}} P(C) \wedge P(C_{N_b}^i) = \max_{C \in C_{N_a} \cup C_{N_b}} P(C)
\]

(8.3)

As in the previous chapters our purpose is to characterize the system behavior by specifying the trace $\rho$ and to investigate the conditions under which the
Figure 8.1. Composition of independent chains with timing constraints.

QoS requirement of the overall system is satisfied. The QoS requirement for the entire system is specified as:

\[ \text{SQoSc}(t, \text{pref}) \overset{def}{=} \text{SQoSci}(t, \text{pref}, f_{q_{N-1}}^a, b_{q_{N-1}}^a, f_{q_{N}}^a, b_{q_{N}}^a, T_N^N, T_N) \land \text{SQoSci}(t, \text{pref}, f_{q_0}^b, b_{q_0}^b, f_{q_1}^b, f_{q_1}^b, T_1^1, T_1). \]

Before we start analyzing the execution of the system we wish to discard those priority assignments to the components in the system which lead to situations in which the QoS requirement is satisfied only by using supplementary buffer space. Hence we analyze first the execution of the system given the following priority assignment to the non time-driven components in the overall system.

\[ P(C_i^b) < P(C_j^a), \forall i, j, C_i^a \in CN_a \setminus C_N^a, C_j^b \in CN_b \setminus C_1^b \quad (8.4) \]

Because the two chains execute on the processor independently of each other and given (8.2) and (8.1) follows that we may apply Theorem 5.4 and Theorem 6.2 for the two chains in the context of the combined system. According to Theorem 5.4, \( CN_b \) reaches its stable phase at the end of the first \( T_1 \). According to Lemma 6.2 and Theorem 6.2, \( CN_a \) reaches its stable phase after all queues in the chain are filled to their capacities. This happens after \( k \geq 2T_N, k \geq 1 \).

When priorities are assigned as in (8.4) all non time-driven components in \( CN_a \) have higher priorities that those in \( CN_b \). That means that during the initial phase of \( CN_a \), \( C_i^b \) will be able to execute (given (8.3)), however the non
time-driven components in $CN_b$ will not be able to execute until all data-driven components in $CN_a$ are blocked - that is during the stable phase of $CN_a$. Given that the stable phase of $CN_a$ occurs only after $k \times 2T_N$, $k \geq 1$ follows that $RSC^b_i$ is not be able to return an empty packet to $C_i$ for $k \times 2T_N$, $k \geq 1$. This means that if $Cap(fq^b_i) < k$ after $Cap(fq^b_i) + 2T_1$, $C^b_i$ becomes blocked at receiving a packet from the input backward queue which implies that the QoS requirement of $CN_b$ is not satisfied hence neither for the overall system. 

The same situation occurs when the priorities are assigned interleaved. In this case only some non time-driven components in $CN_b$ can execute during the initial phase of $CN_a$, while others must wait for the stable phase. Due to the dependencies between components induced by the communication via finite buffers, follows that during $k \times 2T_N$, $k \geq 1$ eventually all non time-driven components will become blocked which leads to the situation that $C_1$ does not receive an empty packet in time and thus the QoS requirement of $CN_b$ is not satisfied again.

We propose a cheaper solution where the buffers do not need to be larger than one position each. Our solution to satisfying the QoS requirement is to control the phasing between the executions of the two systems while restricting the priority assignment of the non time-driven components to:

$$P(C^b_i) > P(C^a_i), \forall i,j, C^a_i \in CN_a \setminus C^a_N, C^b_j \in CN_b \setminus C^b_1$$ (8.5)

In this situation, given (8.3) and Theorem 5.4 follows that $CN_b$ reaches its stable phase before $CN_a$ reaches its own. That means that $CN_a$ reaches its stable phase during the stable phase of $CN_b$. In fact given (8.5) and (8.3) $CN_a$ reaches its stable phase during the idle time of each iteration of the stable phase of $CN_b$.

The phasing ($\Phi$) between the execution of the time-driven components $C^a_N$ and $C^b_i$ is expressed as

$$\Phi = (\sigma(u - t^1_n) - S_\delta(t^1_n)) - (\sigma(s - t^2_k) - S_\delta(t^2_k))$$ (8.6)

where $t^2_k$ is the last iteration of $t^2_k$ before $CN_a$ enters its stable phase ($CN_a$ enters its stable phase between $t^2_k$ and $t^{k+1}_k$). $t^1_n$ is the first iteration of $C^a_N$ during the stable phase of $CN_a$.

In the following theorem we describe the overall execution of the system. We assume here that $S^M_b > T_1$, the $S^M_b \leq T_1$ case is to be solved in the same manner. We show below that when the phasing $\Phi$ satisfies the relations below the QoS requirements of the both chains are satisfied, hence the overall system
QoS requirement is satisfied as well:

\[(S^M_B(t_{C_1}^2)k) < \Phi < T_1) / (T_1 + S^M_B(t_{C_1}^1)k) < \Phi < 2T_1) \quad (8.7)\]

**Theorem 8.1.** Consider a system consisting of CN_d and CN_b, with the priority assignment described in (8.5) and (8.3). Given that \(S^M_B + S^M_b < 2T_1\), \(S^M_b > T_1\), \(S^M_b < 2T_1 - \Phi\) and (8.7) holds, the execution of the system becomes repetitive after a finite prefix:

\[\rho = t_{\text{init}} - (t_{\text{stable}})_{\text{init}}\]

where:

**A. - If** \(S^M_B(t_{C_1}^2)k) < \Phi < T_1\)

1. **If** \(S^M_B < \Phi + T_N - S^M_B(t_{C_2}^1)\) then

\[t_{\text{stable}} = \tau_{C_1}^2 - \tau_{RSC_1}^1 - d(T_0^1 + i * T_1) - \tau_{C_2}^1 - \tau_{RSC_1}^1 - d(T_0^1 + j * T_1) - \tau_{C_1}^1 - \tau_{RSC_1}^1 - d(T_0^1 + T_N) - \tau_{C_2}^1 - \tau_{RSC_1}^1 - d(T_0^1 + (i + 1) * T_N) - \tau_{C_1}^1 - \tau_{RSC_1}^1 - d(T_0^1 + (j + 1) * T_1).\]

**B. - If** \(T_1 + S^M_B(t_{C_1}^1)k) < \Phi < 2T_1\)

1. **If** \(S^M_B \leq \Phi - S^M_B(t_{C_2}^1)\) then

\[t_{\text{stable}} = \tau_{C_1}^2 - \tau_{RSC_1}^1 - d(T_0^1 + i * T_1) - \tau_{C_2}^1 - \tau_{RSC_1}^1 - d(T_0^1 + j * T_1) - \tau_{C_1}^1 - \tau_{RSC_1}^1 - d(T_0^1 + (i + 1) * T_N) - \tau_{C_2}^1 - \tau_{RSC_1}^1 - d(T_0^1 + (j + 1) * T_1).\]

**Figure 8.2. Case A.**
8.2 Composition of chains with timing constraints

\[
\begin{align*}
t^1_{C^0} & \prec t^3_{RSC^1} \prec d(T^0_N + (i+1)T_N) \prec t^2_{C^N} \prec t_{LSC^N} \prec d(T^1_N + (j+1)T_1).
\end{align*}
\]

2. - If \( S^M_b > \Phi - S^M(\frac{t^1_{C^N}}{\Phi}) \) then

\[
\begin{align*}
t_{\text{stable}} = t^2_{C^N} & \prec t^1_{RSC^1} \prec d(T^0_N + i \ast T_N) \prec t^2_{C^N} \prec t^3_{RSC^1} \prec d(T^1_N + j \ast T_1) \prec t^2_{C^N} \prec t^4_{RSC^1} \prec t_{LSC^N} \prec d(T^1_N + (j+1)T_1).
\end{align*}
\]

In both cases A. and B.

- \( t^1_{RSC^1} \prec t^2_{RSC^1} \prec t^3_{RSC^1} \prec t^4_{RSC^1} = t_{RSC^1}^+ \), where \( t_{RSC^1}^+ \) records the cascade execution of components in \( RSC^1_b \).
- \( t^4_{RSC^1} \) is empty in cases A.1 and B.1.
- \( t^1_{LSC^N} \prec t^2_{LSC^N} = t_{LSC^N}^+ \), where \( t_{LSC^N}^+ \) records the cascade execution of components in \( LSC^N_a \).
- \( t^2_{LSC^N} \) is empty in cases A.2, B.1 and B.2.
- \( SQoSci(\rho, \varepsilon, f_{qN-1}?, f_{qN}!, b_{qN}?, b_{qN}!, f_{qN}!, T^N_0, T_N) \) and \( SQoSci(\rho, \varepsilon, f_{q0}?, b_{q0}!, f_{q0}!, f_{q1}!, T^1_0, T_1) \) hold.

Proof. We prove the statement of the theorem by construction of this trace \( \rho \).

We may apply Theorem 5.4 and Theorem 6.2 for the two chains in the context of the combined system although these theorems characterize the execution of the two systems when they execute separately because the two chains are independent of each other and although they share the same processor, \( S^M_a + S^M_b < 2T_1 \) and \( (T_1 = T_N) \) ensure that the conditions of the theorems \( (S^M_a < 2T_N \) and respectively \( S^M_b < 2T_1) \) are satisfied.
We have explained above that according to Theorem 5.4, \( CN_b \) reaches its stable phase during the first \( T_1 \). According to Lemma 6.2 and Theorem 6.2, \( CN_a \) reaches its stable phase after all queues in the chain are filled to their capacities. This happens the earliest during the first \( 2T_0 \). Given (8.3) and (8.7) follows that \( CN_b \) reaches its stable phase before \( CN_a \) reaches its own. That means that \( CN_a \) reaches its stable phase during the stable phase of \( CN_b \). We denote with \( t_{init} \) the state of \( \rho \) that ends just before the next iteration of \( b_{stable} \) interleaved with the first iteration of \( a_{stable} \):

\[
\begin{align*}
t_{init} \uparrow A_a &= t_{init}^a, \\
t_{init} \uparrow A_b &= t_{init}^b - (t_b^k)^{k-1}. 
\end{align*}
\]

According to Theorem 5.4, Lemma 6.2 and Theorem 6.2 at the end of \( t_{init} \), all non time-driven components in \( CN_a \) are blocked at receiving input from the backward queues and all non time-driven components in \( CN_b \) are blocked at receiving input from the forward queues.

We present here the proof for case A.1; proving the statements of the other cases is to be done in the same manner.

As explained above, \( t_{stable} \) starts with the execution of \( t_{Cb}^k \) during which one full packet is produced in \( f_{Cb}^0 \). This causes a de-blocking and execution (given (8.5)) in cascade of components in \( CN_b \) as presented in Theorem 5.4. The trace that records this cascaded execution is denoted with \( t_{RSCb}^1 \). According to the same theorem at the end of \( t_{RSCb}^1 \), all non time-driven components are blocked again at receiving input from their input forward queues. Given that we assume \( S^M_b > T_1 \), the execution of \( t_{RSCa}^1 \) lasts beyond beyond moment \( T_1^l + (2k+1)T_1 \).

Given the priority assignment expressed in (8.5) and (8.3) the only components that can preempt the execution of \( t_{RSCa}^1 \) are \( C^b_1 \) and \( C^a_1 \). Indeed this happens at time \( T_0^N + 2nT_1 \) when \( C_N \) executes \( d(T_0^N + 2nT_N) \sim t_{CN}^1 \), given (8.6) and \( S_b^M(t_{CN}^2) < \Phi < T_1 \). During the execution of \( t_{CN}^1 \) component \( C_N \) releases an empty packet in \( b_{QN-1} \). This causes \( C_{N-1}^N \) to become ready-to-run however \( C_{N-1}^N \) cannot execute yet because there exist still components in \( CN_b \) which are ready-to-run and have higher priorities. Hence the execution of \( C_{N-1}^N \) and in fact the entire cascaded execution of components in \( LSC_{N}^a \) (recorded by \( t_{LSCa}^1 \)) is postponed until the end of \( t_{RSCa}^1 \) when all components in \( RSC_{1}^a \) are blocked again. So far \( t_{stable}^k \) can be expressed as

\[
t_{stable}^k = t_{CN}^k \sim t_{RSCa}^1 \sim d(T_0^N + 2nT_N) \sim t_{CN}^1 \sim t_{RSCa}^2 \sim u.
\]

At moment \( T_0^l + (2k+1)T_1 \), \( C_1^a \) preempts the execution of \( t_{RSCa}^1 \) by executing
8.2 Composition of chains with timing constraints

\[ d(T_0^1 + 2kT_1) - t_{stable}^k. \]

At this point \( t_{stable}^k \) can be expressed as

\[
\begin{align*}
& t_{stable}^k = t_{C_1}^{2^k} - t_{RSC_1}^{1^k} - d(T_0^N + 2nT_N) - t_{C_N^1}^{1} - t_{RSC_1}^{2^k} - d(T_0^1 + 2kT_1) - \\
& t_{C_1}^{1} - t_{RSC_1}^{3^k} - u.
\end{align*}
\]

At the end of \( t_{RSC_1}^{3^k} \) (that is of \( RSC_1^b \)), all components in \( RSC_1^b \) are blocked again and \( C_1^b \) is still blocked from time perspective given that \( t_{RSC_1}^{b^k} < S_b^M \) and \( S_a^M + S_b^M < 2T_1 \). We explained above that the execution of \( LSC_1^b \) has been postponed until the end of \( RSC_1^b \). Sub-trace \( t_{LSC_1}^{b^k} \) is preempted at time \( T_0^N + (2n + 1)T_N \) by \( d(T_0^N + (2n + 1)T_N) - t_{C_N^1}^{1} \). Given \( S_a^M + S_b^M < 2T_1 \) and \( S_b^M < 2T_1 - \Phi \) follows that the execution of \( LSC_1^b \) ends before moment \( T_0^1 + (2k + 1)T_1 \) when \( C_1^b \) becomes ready-to-run again. At the end of \( t_{LSC_1}^{b^k} \) all components in \( LSC_1^b \) have become blocked again at receiving an empty packet from their input backward queues. Given that all other components are blocked as well at the end of \( t_{LSC_1}^{b^k} \), the delay action \( d(T_0^1 + (2k + 1)T_1) \) is executed which advances time until moment \( T_0^1 + (2k + 1)T_1 \). Hence \( t_{stable}^{k} \) can be expressed as

\[
\begin{align*}
& t_{stable}^{k} = t_{C_1}^{2^k} - t_{RSC_1}^{1^k} - d(T_0^N + 2nT_N) - t_{C_N^1}^{1} - t_{RSC_1}^{2^k} - d(T_0^1 + 2kT_1) - \\
& t_{C_1}^{1} - t_{RSC_1}^{3^k} - d(T_0^N + (2n + 1)T_N) - t_{C_N^1}^{2} - t_{LSC_1}^{b^k} - \\
& d(T_0^1 + (2k + 1)T_1).
\end{align*}
\]

At the end of \( t_{stable}^{k} \) the state of all components in the system is identical with their state at the beginning of \( t_{stable}^{k} \). This implies that the execution of the system becomes repetitive after \( t_{init} \) and we can express \( t_{stable} \) in general as

\[
\begin{align*}
& t_{stable} = t_{C_1}^{2^k} - t_{RSC_1}^{1^k} - d(T_0^N + i*T_N) - t_{C_N^1}^{1} - t_{RSC_1}^{2^k} - d(T_0^1 + j*T_1) - \\
& t_{C_1}^{1} - t_{RSC_1}^{3^k} - t_{LSC_1}^{b^k} - d(T_0^1 + (i + 1)*T_N) - t_{C_N^1}^{2} - t_{LSC_1}^{b^k} - \\
& d(T_0^1 + (j + 1)*T_1).
\end{align*}
\]

The fact that the system satisfies the QoS requirements of both chains during \( t_{stable}^{b^k} \) results directly from the execution of the overall system described in the discussion above and illustrated in Figure 8.2. During \( t_{init} \) the QoS requirements of \( CN_a \) and \( CN_b \) are also satisfied. In the case of \( CN_b \) QoS requirements are clearly satisfied given that during its stable phase it only executes \( C_1^b \) which has the highest priority in the system. In the case of \( CN_a \), if the chains were to execute separately we know from Corollary 6.1 that QoS requirement is satisfied. Given \( S_a^M + S_b^M < 2T_1 \) follows that each \( 2T_N = 2T_1 \), \( S_b^M \) is executed at least one time which ensures that \( C_1^a \) has always input. This means that from iteration \( n \) counting back in time \( C_1^a \) is always executed each period \( T_N \). This means that the phasing \( \Phi \) is maintained through all the execution of \( t_{init} \) as of the first execution of \( C_1^a \). Given (8.7) follows that \( C_1^a \) is never preempted by \( C_1^b \).
The next question is how to design a system that satisfies conditions expressed in (8.7). In the following we give an indication for a solution. We propose controlling the start time of the stable phase of chain $CN_a$. We would like to ensure that the stable phase of $CN_a$ starts within the first $2T_N$ from the execution start of $CN_a$. That is to say that $CN_a$ starts executing for the first time within the first $2T_N$ from the execution start of $CN_a$. The following gives one solution to this problem. Indeed, when assigning all buffer capacities only one position, all non-time driven components have priorities assigned ascendingly, and $S^d_a + S^M_b < 2T_N$, all forward queues in $CN_a$ are filled to their capacity and all non-time-driven components in the chain become blocked at receiving input from the backward queues within the first $2T_N$ of the execution of $CN_a$. This means that $CN_a$ reaches its stable phase within its first $2T_N$ of execution. $CN_a$ will become ready-to-run at the end of $t^M_0(t_{LSCN_a})$. To prevent preemption from $C_b^1$ which has a higher priority, the program text of $C_b^1$ could include a delay action $d(\theta)$ meant to delay the execution of $CN_a$ until a time $\theta$. Moment $\theta$ is chosen to be beyond the first period $T_1$ of $C_b^1$ and before the end of the second period $T_1$. This situation corresponds to case A. of Theorem 8.1. Moment $\theta$

![Figure 8.4. Controlling the stable phase start of $CN_a$.](image)

can be chosen such that the execution of the system corresponds to case B., however in this case $CN_a$ would start executing later. In practice case A. is preferred because $CN_a$ produces output earlier.

### 8.3 Summary

We have studied in this chapter composition of independent chains of components sharing and therefore competing for the processor resource. We have shown in the first section that when composing two linear chains composed
of only data-driven components, the chain containing the component with minimum priority overall the composed system, after a finite prefix becomes starved.

In the second section, we present a behavioural analysis of a system composed of two chains with timing constraints. $CN_a$ consists of a component with deferred execution, a component whose behaviour depends on the contents of the input, a number of simple data-driven components and time driven component. This type of chain has been studied in Chapter 6. $CN_b$ is composed of a time-driven component, a number of simple data-driven components and a component with deferred execution. This type of chain has been analyzed in section 5.4. In both cases the time-driven components execute according to the interlacing standard. In practice the first chain corresponds to a video decoding chain and the second chain to a surveillance application that saves on the hard-disk the images captured by the first component.

In our analysis we consider a specific priority assignment to the components in which non time-driven components in $CN_b$ have a higher priority than any component in $CN_a$. Also the time-driven component $C^b_1$ has a higher priority than the time-driven $C^a_N$. We have shown that another priority assignment to the components has negative impact on meeting the QoS requirements of both chains unless more buffer space is allocated. We propose a cheaper solution where the buffers do not need to be larger than one position each. Our solution to satisfying the QoS requirement is to control the phasing between the executions of the two systems while restricting the priority assignment of the non time-driven components as mentioned above. Theorem 8.1 details this solution showing the conditions under which the execution of the system becomes repetitive after a finite prefix and the QoS requirements of the overall system is satisfied.

In conclusion we have shown how to design a system such that the necessary condition for the phasing $\Phi$ is satisfied.
Conclusion

In this thesis we provide a behavioral analysis of real-time systems with interdependent tasks. Our approach to describing the behavior of these systems is incremental. We start with a simple, theoretical case of a linear chain composed of only data-driven components while progressively we increase complexity to realistic systems as found in practice. We close our analysis with composition of systems.

We begin by introducing a number of basic concepts used further on in this thesis. We have formally introduced the systems we study by specifying the syntax for the program text of the system components. This has been achieved by indicating the grammar that generates the component program texts and the grammar for the parallel composition of components.

The semantics associated with a program text is specified using traces. We wish to study the system behavior by focusing on the corresponding trace set. However, before we can characterize the trace(s) that specify the system behavior, we need to identify them. For this purpose we start by considering the trace set containing all arbitrary interleavings of components actions, yielded by the parallel composition of components. On these traces we progressively imposed conditions in the form of predicates. Each time a condition is imposed, it reduces the trace set to a new one, containing only those traces that satisfy the condition. The predicates denote properties or characteristics of
the system execution. They represent the activities of a system executing the programs. In that sense the predicates we impose are mechanisms for the selection of only those traces that specify an execution with the characteristics and properties of our system execution. In the end, the traces that satisfy all conditions specify the system behavior.

Examples of system characteristics specified by predicates include the communication via bounded buffers and the fixed priorities assigned to the components for systems without timing constraints. In the case of systems with timing constraints an additional constraint is imposed to capture the periodic execution of some components.

The conclusion is that both in pipelined systems with and without timing constraints, there exists a unique trace $\rho$ that specifies the actual system execution. A second relevant issue is that there exists a unique eager schedule associated with this trace and this schedule satisfies also the soundness criteria as they have been defined for all time assignments.

Further on, in each of the subsequent chapters we present our model for the dynamic behavior of linear media processing systems composed of different types of components. We consider

- Data-driven components (introduced in Chapter 3),
- Time-driven components (introduced in Chapter 4),
- Components with deferred execution (introduced in Chapter 5),
- Components with execution dependent on the input stream contents (introduced in Chapter 6),
- Demultiplexer components (introduced in Chapter 7).

The behavior of each of the systems we analyze is expressed by means of the unique trace $\rho$ of the actions of the components that make up the chain. We have formally proven that this trace becomes repetitive (the stable phase) after a finite prefix (the initial phase) and we have shown that this trace can be calculated at design time. The trace $\rho$ is completely determined by

- the individual traces of the components determined by their type,
- the timing behavior of components,
- the topology of the system,
- the capacities of the communication buffers,
- static priorities of the components.

The repetitive nature of the unique trace $\rho$ allows at system design time the calculation and optimization of:
Conclusion

- the initial phase,
- start times and response times of the individual tasks,
- response times of a complete chain of components,
- number of context switches, and the position of the context switches in the component traces, which is an indicator for their overhead cost,
- capacities for each buffer in the system,
- idle times during the system execution.

In the case of systems with timing constraints we prove that a time-driven component has the same influence on the overall execution of a chain as a data-driven component with minimum priority has on a chain composed of only data-driven components. This reduces part of the analysis of time-driven systems to be identical to that of the data-driven system in Chapter 3. This result is used in the analysis of systems with timing constraints presented in Chapters 4-8. Additionally, for each of the systems with timing constraints we present techniques that guarantee satisfying QoS requirements with respect to the frame rate QoS metric. One other important result of Chapter 4 refers to CPU overload situations in which the time-driven component at the end of a chain misses its deadline for a number of periods. In these cases we show how to design the system such that there always exists an infinite suffix of the trace $\rho$ during which the chain satisfies the QoS requirements. The results of this analysis are relevant because they show a cheap solution at design time of systems that guarantees meeting QoS requirements for an infinite suffix of the system trace. The solution is suitable for systems that experience high variations in computation times of tasks and it concerns trading off small additional amounts of memory in a specific buffer for much lower processing power.

The repetitive nature of systems is an important property that also makes reasoning about composition of dependent and independent sub-systems much easier. Designers need only to reason in terms of patterns of execution at the level of the system instead of reasoning about the individual behaviors of components within the whole system. This approach also makes systems compositional in the sense that the effect of inserting (or withdrawing) components from a chain can be rigorously predicted and controlled.

This is shown in the advanced stages of our analysis where we tackle composition of systems studied previously. In Chapter 7 we study the execution of a system consisting of a composition of dependent chains studied in Chapter 6. The main difference between the system studied here and those studied in all previous chapters is the system topology: in the previous chapters we have studied linear chains while in Chapter 7 we tackle the analysis of a system with
branched topology. Our approach is to characterize the execution of each composing sub-chain within the overall execution of the system, and subsequently we characterize the interleaving of these two executions while pointing out the situations in which the QoS requirements are satisfied. In characterizing the individual execution of each sub-chain within the overall system execution we use a similar approach as presented in Chapter 6. We explain that the actual interleaving between the executions of the two composing sub-chains is determined by the ratio between the periods of the two time-driven components at the end of the two sub-chains, the contents of the input stream which influences the computation times of the trace actions, the duration of the deferral times of the first component and the priority assignment of the components. Practical applications concerning QoS, optimizing system resources and timing properties are addressed again at the end of the chapter.

In Chapter 8 we analyze the behavior of systems composed of two independent linear chains as opposed to Chapter 7 where we studied the composition of two dependent chains. We tackled two types of independent composition: where none of the chains have timing constraints and the situation where both chains have timing constraints. In the first case both chains are composed of only data-driven components. We show that composing these chains is not advisable because after a finite prefix one of the chains becomes starved. In the second case the first chain corresponds to a video decoding chain and the second chain to a surveillance application that saves on the hard-disk the images captured by the first component. The challenge in this case is to find solutions for designing the composition of the chains such that both chains satisfy their QoS requirements. We show that certain priority assignments imply supplementing the buffer capacities in the chains which is costly. We propose and detail a cheaper solution in which the buffers do not need to be larger than one position each. Our solution to satisfying the QoS requirement is to impose a specific priority assignment to the components and to control the phasing between the executions of the two systems. We also show how to design a system such that the necessary condition for the phasing is satisfied.

In conclusion we have presented in this thesis an approach for predicting and controlling at system design time the overall behavior of real-time component-based systems built according to the Pipes and Filters architectural style and scheduled using fixed priority scheduling. In that sense we have been able to successfully address the initial problem of building media processing systems that satisfy QoS requirements while using a minimum of resources. Our goal was not to present an exhaustive behavioral analysis of these systems but to thoroughly explain the approach to take in tackling the challenges associated with predicting and controlling their execution. Future work could
include analyzing the behaviour of systems composed of two dependent sub-chains joined by means of a mixer component. Another important direction in which the present work can be extended is to analyze the behavior of these systems when executing on multi-processor platforms.
Bibliography


BURNS, A. [1994], Preemptive priority based scheduling: An appropriate

**Buschmann, F. and R. Meunier et al. [1996]**, *Pattern-Oriented Software Architecture*, John Wiley & Sons Ltd.


**Foster, I., A. Roy, and V. Sander [2000]**, A Quality of Service architecture that combines resource reservations and application adaptation, *Proc. 8th International Workshop on Quality of Service (IWQoS)*, 181–188.


**Hentschel, C., R.J. Bril, Y. Chen, R. Braspennin g, and T.-H. Lan**


PENG, S. [2001], Complexity scalable video decoding via IDCT data pruning, *Digest of technical papers IEEE International Conference on Consumer Electronics (ICCE)*, 74–75.
PHILIPS, SEMICONDUCTORS. [1999], *Philips TriMedia™ Documentation Set*, SDE 2.1.
TANENBAUM, A.S. [2003], *Computer Networks*, Pearson Education.


Publications


To be submitted

- M.A. Weffers-Albu, J.J. Lukkien; Behavioral Analysis of a Video Decoding Chain; Real-Time Systems Symposium.
• M.A. Weffers-Albu, J.J. Lukkien; Behavioral Analysis of a Chain Composition; IEEE Transactions on Computers Journal.
Symbol Index

Operators

- $|t|$ the length of a trace $t$. 28
- $s \cdot t$ concatenation of traces $s$ and $t$. 28
- $s \subseteq t$ $s$ is prefix of $t$. 28
- $t \uparrow A$ projection of a trace $t$ to a certain alphabet $A$ returns the trace obtained from $t$ by removing all symbols not in $A$ while maintaining the order given in $t$. 32
- $(t,a)$ the counter operator returns the number of occurrences of action $a$ in trace $t$. 32
- $C_0 || C_1$ parallel composition of two component programs $C_0$ and $C_1$. 32
- $C_0 b_i [in \ s \ of \ T]$ $C_0$ is blocked at $b_i$ in state $s$ of $T$. 34
- $C_N b_t a [in \ s \ of \ T_{\sigma_c}]$ $C_N$ is blocked from time perspective at $a$ in state $s$ of $T_{\sigma_c}$. 47
- $Q?$ receive action via a queue $Q$. 29
- $Q!$ send action via a queue $Q$. 29

Greek Symbols

- $\varepsilon$ the empty trace. 28
- $\delta_p(a,k)$ function returning the computation time of the $k^{th}$ occurrence of action $a$ in a trace. 43
- $\delta(a^k)$ short-hand notation for $\delta_p(a,k)$. 44
- $\Phi$ phasing between the execution of $C_N^\phi$ and $C_1^\phi$. 185
- $\mu$ maximum delay at preemption of a component by a time-driven component. $\mu$ is required to be at most the maximum over all computation times of atomic actions of all components in the system. 42
- $\rho$ unique trace of $T_{pc}$. 39
- $\sigma(s)$ schedule function returning for any state $s$. 207
from $St(T_U)$, the finishing time of state $s$. 44

$\sigma_{eager}(s)$ function returning for every state $s$ of a trace $t$ the minimum of the values returned by all schedules (in $S_\sigma$) for state $s \in St(T_U)$. 45

$\tau_k$ length of deferral time $\tau_k$. 111

$A$

$a, b, c, d$ notations for trace actions.

d$^k$ the $k^{th}$ occurrence of an action $a \in A(C_i)$ in a trace. 43

$A$

the union of the alphabets of all components ($A = \bigcup_{i=1..N} A'(C_i)$). 35

$A^\omega$ the set of all infinite traces which are formed from actions in the set $A$. 35

$A(C)$ alphabeth of a program $C$. 24

$A'(C)$ the set of atomic actions appearing in a program trace. 28

$Aalph$ function that for each basic statement in alphabet $A(C)$ returns the corresponding set of atomic actions that appear in the program trace. 28

$B$

$B$ basic statement. 24

$B(s)$ the set of components into blocked components in state $s$. 34

$C$

$C$ component program. 24

$C_m$ the component with minimum priority in the chain. 53

$Cap(Q)$ capacity of queue $Q$. 25

$Comp$ function taking as argument an action and returning the component with the alphabet to which the action belongs. 34

$CompTr_i$ component trace that records the actual execution of $C_i$. 31

$CN$ the set of components in a chain composed of $C_1, C_2, \ldots, C_N$. 67

$CS$ compound statement. 24
Symbol Index

E
EXPR expression. 24

F
fv minimum number of packets that make up an encoded frame belonging to the video elementary stream. 161
Fv maximum number of packets that make up an encoded frame belonging to the video elementary stream. 161
fa minimum number of packets that make up an encoded frame belonging to the audio elementary stream. 161
Fa maximum number of packets that make up an encoded frame belonging to the audio elementary stream. 161
flush flush time. 66

G
G guard. 24

L
L(\(Q\)) length of \(Q\), where the length expresses the number of elements currently stored in \(Q\). 25
LSC_i the set of components preceding component \(C_i\) in a chain. 67

N
NCS function taking as argument a finite trace from a trace set \(T\), and returning the number of context switches occurring in the trace. 34
NUMBER numeric constant. 24

P
P priority function that returns for each component a unique natural number with the interpretation that a higher number means a higher priority. 38
Pref(\(t\)) the set of all prefixes of \(t\). 28

R
\textbf{RR}(s) \quad \text{the set of components into ready-to-run components in state } s. 
\textbf{RSC}_i \quad \text{the set of components following component } C_i \text{ in a chain.} 
\textbf{RTC} \quad \text{response time of the chain function that returns for a packet } k \text{ as the time counted from the moment that the packet starts being processed by the first action of component } C_1 (f_{q0}) \text{ until the finish time of last action of } C_N \text{ that processes } k (f_{N!}). 

\textbf{S} 
\begin{align*}
s, t, u, v, w & \quad \text{notations for traces.} 
S & \quad \text{processing time between the production of two consecutive packets in the case where all trace actions do not have variable computation times.} 
S^k & \quad \text{processing time between the production of two consecutive packets } k - 1 \text{ and } k \text{ in } f_{q(k-1)}. 
S^M & \quad \text{maximum processing time between the production of two consecutive packets.} 
S^k_a & \quad \text{the processing time between the production of two consecutive packets } k \text{ and } k - 1 \text{ in } f_{q(k-1)}. 
S^M_a & \quad \text{maximum over all values } S^k. 
S^M_b & \quad \text{maximum over all values } S^k. 
SC & \quad \text{sequential composition.} 
SC_v & \quad \text{video decoding sub-chain of the branched system.} 
SC_a & \quad \text{audio decoding sub-chain of the branched system.} 
S^M_v & \quad \text{maximum processing time between the production of two consecutive packets in } f_{q(k-1)} \text{ including the interfering of audio processing.} 
S^M_a & \quad \text{maximum processing time between the production of two consecutive packets in } f_{q(k-1)} \text{ including the interfering of video processing.} 
S^M_{pv} & \quad \text{maximum processing time between the}
production of two consecutive packets in $fq^N$ not including the interfering, interleaved processing of the audio sub-chain.

*Symbol Index*

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{pa}^{M}$</td>
<td>maximum processing time between the production of two consecutive packets in $fq^M$ not including the interfering, interleaved processing of the video sub-chain.</td>
</tr>
<tr>
<td>$St(T)$</td>
<td>the set of states (prefixes) during execution characterized by the prefixes of the trace set $T$.</td>
</tr>
<tr>
<td>$SQoS_c$</td>
<td>predicate specifying the QoS requirement of a time-driven component that does not execute according to the interlacing standard.</td>
</tr>
<tr>
<td>$SQoS_{overload}$</td>
<td>predicate specifying the QoS requirement for overload situations.</td>
</tr>
<tr>
<td>$SQoS_i$</td>
<td>predicate specifying the QoS requirement of a time-driven component that executes according to the interlacing standard.</td>
</tr>
<tr>
<td>$Sw(s,a)$</td>
<td>function returning the time to switch context in state $s$ to $Comp(a)$.</td>
</tr>
<tr>
<td>$S_{\delta}(t)$</td>
<td>function that returns the sum of computation times of actions in trace $t$.</td>
</tr>
<tr>
<td>$S_{\delta}^{M}(t)$</td>
<td>sum of the worst case computation times of actions in trace $t$.</td>
</tr>
<tr>
<td>$S_{\sigma}$</td>
<td>set of schedule functions of the concurrent execution of components $C_i$, $i = 1..N$ on a processor.</td>
</tr>
<tr>
<td>$T$</td>
<td>trace that records one iteration of $CompTr_i$.</td>
</tr>
<tr>
<td>$t_{C_i}^{1}$</td>
<td>trace corresponding to the odd executions of $C_i$.</td>
</tr>
<tr>
<td>$t_{C_i}^{2}$</td>
<td>trace corresponding to the even executions of $C_i$.</td>
</tr>
<tr>
<td>$t_{C_N}^{1}$</td>
<td>trace corresponding to the odd executions of $C_N$.</td>
</tr>
<tr>
<td>$t_{C_N}^{2}$</td>
<td>trace corresponding to the even executions of $C_N$.</td>
</tr>
<tr>
<td>$t_{init}$</td>
<td>initial phase.</td>
</tr>
</tbody>
</table>
$t_{stable}$: stable phase.

$T_0^1$: execution start time of component $C_1$.

$T_1$: period of component $C_1$.

$T_0^n$: execution start time of component $C_n$.

$T^n$: period of component $C_n$.

$T_{il}$: the set of traces that results from the interleaving of the component traces.

$T_{cc}$: set containing channel-consistent traces.

$T_{pc}$: set containing priority-consistent traces.

$T_{sc}$: set containing schedule-consistent traces.

$Tr$: function that takes as input a program $C$ and returns the set of traces that are possible according to the program $(Tr(C) \subseteq Traces(A'(C)))$.

$Traces(X)$: the set of all finite and infinite sequences over a set $X$.

$V$: variable.

$VAR$: variable.

$X$: variable.

$X_a$: maximum number of consecutive packets of the audio type in the stream.

$X_v$: maximum number of consecutive packets of the video type in the stream.
Summary

The analysis presented in this thesis considers the problem of processing a media stream by a system consisting of a chain of given off-the-shelf software components, executed on a scarce-resource embedded platform. Each component corresponds to a task, and the communication between tasks is buffered. The essential requirement on the physical platform is cost-effectiveness, and the requirement on the system is robustness. These requirements lead to minimizing the resources made available to the system to the limit that it remains robust. In the context of this work the robustness criteria for a system are meeting the system real-time constraints. The real-time constraints come from the fact that media processing systems must display audio/video information at a certain rate in order to avoid audio/video artefacts. This implies that the degree in which the real-time constraints are met directly influences the quality of service provided by the system. To ensure that the timing constraints are met, the chain is provided with a guaranteed resource budget. Within the chain, the tasks are scheduled using fixed priority scheduling. Due to the dependencies between the tasks, and their different behaviors, it is difficult to predict the behavior of the chain. Hence, it is difficult to determine the minimum needed resource budget, to predict chain response time, to minimize buffer sizes and context switch overhead, and to reason about chain composition.

The current practice in the domain of media processing systems lacks a theoretical underpinning that helps designers and developers beyond intuition and experience. Such a theory is also needed to control the chain behavior at design time, to make sure timing requirements are met even in overload situations. This thesis provides an underlying theory that helps engineers to reason rigorously about system behavior and associated resource needs. It starts from the experimental observation that, under certain conditions, a media processing chain assumes a repetitive behavior, the stable phase, after a finite initial phase. Starting from this observation a theoretical model for the execution of streaming chains in media processing systems is built. The general strategy is to analyze streaming systems in an incremental manner starting from a simple theoretical case, to realistic streaming chains that include branching and more complex types of components.
The approach allows calculating the execution order of the components in a chain, expressed as a trace of actions taken by each component. The analysis formally proves that the behavior of the chain can be expressed as a unique trace, which, under certain conditions imposed at chain design time, assumes a repetitive pattern after a finite prefix. The trace is completely determined by the individual traces of the components, the computation times of the component actions, the topology of the chain, the capacities of the communication buffers, and the static priorities of the components. Given the computation times for each action in the trace, the associated schedule can be derived. The unique trace of actions and the schedule prove an excellent starting point for further analysis. Start times and response times of the individual components and the complete chain are immediately available. The number of context switches, and the position of the context switches in the component traces, which is an indicator for their overhead cost, can be extracted from the trace. Aside of that, the theory provides corollaries showing how to design the system such that each of the above parameters can be optimized. Also, given the individual traces of the components and the channel constraints, the minimum necessary and sufficient capacities for each buffer in the chain are calculated. Finally, the analysis shows the conditions to be imposed at design time such that even in overload situations the chain satisfies (during an infinite suffix of the unique trace) its real-time constraints that influence directly the quality of service provided.
Curriculum Vitae

Alina Albu finished her studies at the West University of Timisoara, Computer Science division of its Faculty of Mathematics in 1996. From 1996 until 2000 she worked as a software designer and project leader in various software companies. In 2000 she joined the post-graduate (post-M.Sc.) Software Technology education and training programme of the Stan Ackermans Institute of the Eindhoven University of Technology. She graduated in 2003 and received her Professional Doctorate in Engineering (PDEng) degree. In July 2003, she started working as a PhD-candidate at the Department of Mathematics and Computer Science (Group: System Architecture and Networks) of the Eindhoven University of Technology. She carried out her project on the subject of Quality-of-Service in In-Home Digital Networks at the Philips Research Laboratories (Group: Information Processing Architectures) in Eindhoven. Currently she works as a research scientist in the Experience Procesing group of the Philips Research Laboratories Eindhoven.