Timing Architecture for ESS

Autor: Javier Cereijo García

2020

Directores: Daniel Piso Fernández
Roberto Rodríguez Osorio

Tutor: Roberto Rodríguez Osorio

Programa de doctorado en Investigación en Tecnologías de la Información

UNIVERSIDADE DA CORUÑA

Dr. Daniel Piso Fernández
Director Engineering
Architecture Technology Group
Arm Ltd.

Dr. Roberto Rodríguez Osorio
Profesor Titular de Universidad
Dpt. de Ingeniería de Computadores
Universidade da Coruña

CERTIFICAN

Que la memoria titulada “Timing Architecture for ESS“ ha sido realizada por D. Javier Cereijo García bajo nuestra dirección en el Programa de Doctorado Interuniversitario en Investigación en Tecnologías de la Información, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor.

En A Coruña, a 10 de Julio de 2020

Fdo.: Daniel Piso Fernández
Director de la Tesis Doctoral

Fdo.: Roberto Rodríguez Osorio
Director de la Tesis Doctoral

Fdo.: Javier Cereijo García
Autor de la Tesis Doctoral

Acknowledgements

I would like to acknowledge and thank a lot of people without whom I would not have been able to do this thesis.
In the first place I would like to thank my supervisors for all the support and help while doing this thesis. Without them I would not have achieved it. Daniel Piso Fernández who was my key and door to the European Spallation Source, and Roberto Rodríguez Osorio who helped me at the university and with the research for this thesis.
I would also like to thank Javier Díaz Bruguera, who introduced and gave me the opportunity to do this thesis at the European Spallation Source.

I would like thank all my colleagues at the European Spallation Source, in the Integrated Control System division and specially those of them in the Hardware and Integration group, including my line manager Karl Vestin and work package manager Faye Chicken. In no particular order, I am grateful to Joao Paulo Martins, Saeed Haghtalab, John Sparger and Nicklas Holmberg for challenging me with new usecases for the timing system. This gratefulness also includes the people that are not part of the European Spallation Source any more, like Nick Levchenko and David Brodrick for sharing their integration experience, Ursa Rojec for introducing me to EPICS and Simone Farina for his help with everything and specially hardware. Of course I would like to thank Jerzy Jamroz and Felipe Torres González for their discussions of different implementations of timing systems and Michael Davidsaver for his help with *mrfioc2*, EPICS and everything related to timing in general. I would also like to thank the stake holders of the timing system for their feedback and suggestions. Also Jukka Pietarinen from Micro-Research Finland.

I would specially like to thank the chief engineer Timo Korhonen for being my supervisor at the European Spallation Source and for the help, guidance and discussions about all the technologies that I have used when

working with the timing system. And finally I would like to thank the person at the European Spallation Source who helped me the most when I was not sure how to advance and who pushed me to continue and finish the job. Thank you Jeong Han Lee.

There is more people that have helped me with this project, although I have not mentioned. I would like to thank all of them.

Finally I would like to thank my friends, family and my girlfriend M-C for their support and listening to me.

Resumo

O sistema de temporización é unha compoñente fundamental para o control e sincronización de instalacións industriais e científicas, coma aceleradores de partículas. Nesta tese traballamos na especificación e desenvolvemento do sistema de temporización para a European Spallation Source (ESS), a maior fonte de neutróns actualmente en construción. Abordamos este traballo a dous niveis: a especificación do sistema de temporización, e a implementación física de sistemas de control empregando circuítos reconfigurables.

Con respecto á especificación do sistema de temporización, deseñamos e implementamos a configuración do protocolo de temporización para cumprir cos requirimentos do ESS e ideamos un modo de operación e unha aplicación para a configuración e control do sistema de temporización.

Tamén presentamos unha ferramenta e unha metodoloxía para implementar sistemas de control empregando FPGAs, coma os nodos do sistema de temporización. ámbalas dúas están baseadas en statecharts, unha representación gráfica de sistemas que expande o concepto de máquinas de estados finitos, orientada a sistemas que necesitan ser reconfigurados rápidamente en múltiples localizacións minimizando a posibilidade de erros. A ferramenta crea automaticamente código VHDL sintetizable a partir do statechart do sistema. A metodoloxía explica o procedemento para implementar o statechart como unha arquitectura microprogramada en FPGAs.

Resumen

El sistema de temporización es un componente fundamental para el control y sincronización de instalaciones industriales y científicas, como aceleradores de partículas. En esta tesis trabajamos en la especificación y desarrollo del sistema de temporización para la European Spallation Source (ESS), la mayor fuente de neutrones actualmente en construcción. Abordamos este trabajo en dos niveles: la especificación del sistema de temporización, y la implementación física de sistemas de control empleando circuitos reconfigurables.

Con respecto a la especificación del sistema de temporización, diseñamos e implementamos la configuración del protocolo de temporización para cumplir con los requisitos de ESS e ideamos un modo de operación y una aplicación para la configuración y control del sistema de temporización.

También presentamos una herramienta y una metodología para implementar sistemas de control empleando FPGAs, como los nodos del sistema de temporización. Ambas están basadas en statecharts, una representación gráfica de sistemas que expande el concepto de máquinas de estados finitos, orientada a sistemas que necesitan ser reconfigurados rápidamente en múltiples localizaciones minimizando la posibilidad de errores. La herramienta crea automáticamente código VHDL sintetizable a partir del statechart del sistema. La metodología explica el procedimiento para implementar el statechart como una arquitectura microprogramada en FPGAs.

Abstract

The timing system is a key component for the control and synchronization of industrial and scientific facilities, such as particle accelerators. In this thesis we tackle the specification and development of the timing system for the European Spallation Source (ESS), the largest neutron source currently in construction. We approach this work at two levels: the specification of the timing system and the physical implementation of control systems using reconfigurable hardware.

Regarding the specification of the timing system, we designed and implemented the configuration of the timing protocol to fulfil the requirements of ESS and devised an operation mode and an application for the configuration and control of the timing system.

We also present one tool and one methodology to implement control systems using FPGAs, such as the nodes of the timing system. Both are based on statecharts, a graphical representation of systems that expand the concepts of Finite State Machines, targeted at systems that need to be reconfigured quickly in multiple locations minimizing the chance of errors. The tool automatically creates synthesizable VHDL code from a statechart of the system. The methodology explains the procedure to implement the statechart as a microprogrammed architecture in FPGAs.

Preface

Industrial and scientific facilities are getting more and more complex to deal with new challenges and become more efficient. This affects the equipment that makes up these centres but also, and maybe even more importantly, the systems that integrate all of these devices so that they can work together to fulfil the goal of the facility. One of these is the control system, which is responsible for managing the rest of the systems so that they operate in the correct way and in unison. Control systems are intricate networks that comprise hardware, software and their respective configurations, and that are integrated together to successfully run the machine or facility.

One of the integral parts of control systems is the timing system. The timing system is in charge of synchronizing all of the devices together and to the rest of the world outside the facility or centre. This last part is usually done by connecting to an external source, such as the Global Positioning System, from which the timing system derives its reference of time and that it uses to orchestrate the facility. Timing systems usually have a master node that defines the time for the centre disciplined by the external source and dispatches it to the rest of the timing system, formed by slave nodes. The master node is also the central point where the operation and configuration are managed from, and it codes this configuration so that it can be sent and used by the slave nodes. The slave nodes receive the time and information sent by the master node and react accordingly. The typical reactions are sending precise triggers and timestamping different signals and occurrences.

There are a number of challenges that the timing system needs to solve for performing its duties correctly. The most important one is the synchronization of the facility. For achieving synchronization it is mandatory to define one common master clock and send it to the rest of the timing system nodes. If this is not done, and it is decided for each of the nodes to run its own clock, all of them with the same frequency, one issue quickly

shows up: even if the clocks are off by just a fraction of a part per million, given the frequency at which the timing system operates, usually around 100 MHz, in a matter of a few seconds the nodes will drift from each other by more than one cycle of the clock. The only solution is then to define and distribute a common master clock that is shared by all of the nodes. On top of defining the common clock, it is needed to define a moment in time that acts as the reference of time and distribute this moment to all nodes without delay. Since this is not possible, specially because the nodes that need to be synchronized are far away from each other, in some cases hundreds of meters or kilometres away, an alternative solution is needed, which entails calculating the delay of the transmission between the nodes, so that the original moment in time can be calculated.

In this thesis we study, integrate and make some contributions to the timing system of one scientific facility currently under design and construction, the European Spallation Source. The European Spallation Source is a scientific facility for research with neutrons that will be the largest in the world when it starts operating. It is formed by a linear accelerator that shoots a beam of protons at a target wheel that produces neutrons by the spallation process. The neutrons are then guided to a set of experiments, where they are used to perform science in different fields. The European Spallation Source is a collaboration among several European countries, that design and build the different parts and systems all over Europe as an in-kind project, and deliver the parts to Lund, in southern Sweden, where they are assembled together. While the actual facility is located in Lund, the data produced by the experiments will travel to Copenhagen in the neighbour country of Denmark where it will be processed and stored.

The timing system at the European Spallation source is implemented with a master node, called a event generator, that sends events and other timing information to event receivers, which react to the events as they are configured to, mainly to trigger the different devices in a synchronized way. In the context of the timing system, an event is an enumerated pulsed signal. Most of the nodes of the timing system at the European Spallation Source are provided by Micro-Research Finland. This timing system is in charge of creating and distributing the events that are used to trigger the different devices, distributing synchronous clocks, defining and sharing a common time reference used for timestamping signals and other occurrences and distributing some beam-related parameters throughout the facility.

In this thesis we deal with the design, implementation and integration of the timing system for the European Spallation Source. This includes studying the European Spallation Source facility, the systems and devices that form it and their requirements. Then we come up with a way of implementing all of these in the protocol that is used by the timing system, and we develop a strategy for implementing the control and configuration of the event generator and the event receivers. This includes developing the lists of events and data distributed by the timing system and the configuration of the nodes according to these lists.

On top of this we present some contributions that we developed for the timing system of the European Spallation Source. Among them is a new mode of operation of the event receivers, so that they can work independently from an event generator. This has been very useful at the European Spallation Source due to the in-kind nature of the project. Another contribution relates to a high-level application that will be used for commissioning and ramp-up of the European Spallation Source after shutdown periods that allows to pre-define a series of sequences of events and data that are sent automatically and in a synchronized way. We also present a new board devised and designed by the European Spallation Source and an in-kind collaborator, and that will be used to deliver timing in locations of the facility where, for a number of reasons such as constrained space or just saving costs, it is impossible or at least impractical to deploy one of the normal event receivers. Finally we present a tool and a methodology to implement control systems in reconfigurable hardware minimizing the chance of errors. Both the tool and the methodology are based on statecharts, an expansion of Finite State machines. The tool translates the system represented by an statechart into VHDL ready to be synthesized and implemented in hardware, while the methodology explains how to create a microprogrammed architecture targeted at FPGAs that implements statecharts. Having tools and processes that are error-free is very important in big facilities such as the European Spallation Source, since the complexity of the machine induces the appearance of issues that may prevent the facility from working properly or even be destroyed or harm people.

Contents

List of Figures

List of Tables

List of abbreviations

Abbreviation	Definition
AC	Alternating Current
AMC	Advanced Mezzanine Card
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Processor
BCM	Beam Current Monitor
CA	Channel Access
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
dBm	Decibel-milliwatt
DIN	Deutsches Institut für Normung (German organization for standardization)
DMSC	Data Management and Software Centre
DOM	Document Object Model
DTL	Drift Tube Linac
EPICS	Experimental Physics and Industrial Control System
ERIC	European Research Infrastructure Consortium
ESS	European Spallation Source
EVG	EVent Generator
EVM	EVent Master
EVR	EVent Receiver
FC	Faraday Cup
FEL	Free Electron Laser
FIFO	First-In-First-Out
FMC	FPGA Mezzanine Card
F-O	Fan-Out
FPGA	Field-Programmable Gate Array
FRIB	Facility for Rare Isotope Beams
FSM	Finite State Machine

Abbreviation	Definition
FW	FirmWare
Gbps	Giga bit per second
GeV	Giga electron-volt
GPS	Global Positioning System
GSI	GSI Helmholtz Centre for Heavy Ion Research
HDL	Hardware Description Language
HEBT	High Energy Beam Transport
Hz	Hertz
IBM	International Business Machines corporation
ICS	Integrated Control System
ID	IDentifier
IFB	InterFace Board
ILL	Institut Laue-Langevin
I/O	Input/Output
IOC	Input/Output Controller
IT	Information Technology
J-PACR	Japan Proton Accelerator Research Complex
JTAG	Joint Test Action Group
KiB	Kibi Byte
LCLS	Linac Coherent Light Source
LEBT	Low Energy Beam Transport
LED	Light-Emitting Diode
LLRF	low Level Radio Frequency
LPC FMC	Low Pin Count FPGA Mezzanine Card
LPF	Low Pass Filter
LPS	Local Protection System
LUT	Look-Up table
LVPECL	Low-Voltage Positive Emitter-Coupled Logic
mA	milli Ampere
MByte	Mega Byte
MCH	MicroTCA Carrier Hub
MEBT	Medium Energy Beam Transport
MeV	Mega electron-volt
MHz	Mega Hertz
MPS	Machine Protection System
MRF	Micro-Research Finland
ms	milli second
mTCA	Micro Telecommunications Computing Architecture
microTCA	Check mTCA

Abbreviation	Definition
MW	Mega Watt
ns	nano second
NTP	Network Time Protocol
PBI	Proton Beam Instrumentation
PC	Personal Computer *or* Program Counter
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect express
PD	Phase Detector
PL	Programmable Logic
PLA	Programmable Logic Array
PLC	Programmable Logic Controller
PLL	Phase Locked Loop
PPS	Pulse Per Second
PS	Processing System
PTP	Precision Time Protocol
PV	Process Variable
RAM	Random Access Memory
RF	Radio Frequency
RFQ	Radio Frequency Quadrupole
RMS	Root Mean Square
RTM	Rear Transition Module
SCSI	Small Computer System Interface
SFP	Small Form-factor Pluggable
SLS	Swiss Light Source
SNS	Spallation Neutron Source
SoC	System-on-Chip
TTL	Transistor-Transistor Logic
UK	United Kingdom
UML	Unified Modeling Language
USA	United States of America
UTC	Coordinated Universal Time
μTCA	Check mTCA
VCO	Voltage-Controlled Oscillator
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VME	Versa Module Eurocard
XML	Extensible Markup Language

Table 1: List of abbreviation used in this thesis.

Chapter 1

Introduction

Making new science and developing better engineering practises is becoming more and more complex each day, with better, bigger, more powerful and more efficient experiments and machines being needed to reach new grounds, as current ones are reaching their limits. One example is the biggest machine in the world, the Large Hadron Collider at CERN, the European Organization for Nuclear Research located in Geneva, which intends to go beyond our current understanding of Physics and for that it needs more energetic collisions, and thus a larger diameter to minimize synchrotron radiation [3]. Other example is the fusion nuclear reactor ITER in France, which intends to be the first fusion device to produce a ten-fold return on energy (Q=10) [4] and thus be efficient enough for commercial use.

These industrial and scientific facilities require for their operation reliable, state-of-the-art control systems, which get more sophisticated as the complexity of the systems or devices they manage increases. These control systems are intricate networks that comprise hardware, software and their respective configurations, and that are integrated together to successfully run the machine or facility. It is of uttermost importance that all configurations are implemented in a flexible and error-free manner, since the increasing sophistication of the control systems brings out any issue or defect with harmful consequences. For this reason it is becoming more and more prevalent the use of tools that automatize and simplify the process of designing, implementing and integrating control systems, providing safer, more reliable and more flexible systems for better industrial and scientific facilities.

(a) (b) (c)

Figure 1.1: (a) Buddha sculpture, (b) X-ray image, (c) neutron image. Source [1].

1.1 Neutrons for science

Neutrons are a very powerful and extensively used tool for researching and understanding the internal structure of matter and the properties and composition of materials. Neutrons have some special characteristics, such as the possibility of matching their energy and wavelength scale to atomic and molecular processes, or using them as weakly coupled probes, as they can penetrate deeply and harmlessly in the sample being studied. They are also very sensitive to hydrogen, allowing the study of the structure and dynamics of organic matter, such as polymers and biological matter [5]. Usually a beam of neutrons is used to probe a small sample under study. The sample scatters the neutron beam in a specific way that allows scientist to obtain information, for example the scattering pattern can produce an image of the atomic structure. Among the applications of neutron scattering are clean energy and the environment, pharmaceuticals, healthcare, nanotechnology, materials engineering, fundamental physics, IT, biochemical engineering, food science, drug synthesis and biophysics [6]. Figure 1.1 shows a Buddha sculpture studied under two different techniques: X-ray and neutrons, where it is possible to notice the difference in the output between the two methods.

There are two main kinds of neutron sources: continuous and pulsed sources. The most common type of continuous sources are nuclear reactors that produce a continuous flux of neutrons, such as the Institut Laue-

Langevin (ILL) [7] in Grenoble, France. Pulsed sources, on the other hand, trigger the emission of neutrons from a target at a certain repetition rate. Spallation sources, which are pulsed sourced, use a high power proton beam, exceeding 1 MW, to hit a metal target and destabilise its atoms, which then emit neutrons. Examples of spallation sources are Spallation Neutron Source (SNS) [8] in Oak Ridge National Laboratory (USA), Japan Proton Accelerator Research Complex (J-PARC) [9], or the European Spallation Source (ESS) in Sweden.

1.1.1 The European Spallation Source ERIC

ESS is an European Research Infrastructure Consortium (ERIC) with 13 European members, including the host nations of Sweden (where the ESS facility is being built) and Denmark (where the Data Management and Software Centre (DMSC) will be located) [10]. The construction of ESS started in 2014 in Lund, southern Sweden, and should be operational at full performance in 2025 [11]. The production of neutrons happens thanks to two big different parts:

- A proton linear accelerator, which will be the most powerful linear proton accelerator ever built [10]. It will produce a 5 MW, 2.86 ms long beam of protons at 2 GeV, which will hit the target with a repetition rate of 14 Hz. The average pulse current is 62.5 mA [11]. It will have a normal conducting section consisting of a Radio Frequency Quadrupole (RFQ) and a Drift Tube Linac (DTL) that accelerates the beam produced by the proton source, with a Low Energy Beam Transport (LEBT) and Medium Energy Beam Transport (MEBT) in between the different systems. The normal conducting section will accelerate the beam up to 90 MeV, most of the acceleration will be achieved in superconducting cavities: spoke cavities up to 200 MeV, medium-β cavities up to 570 MeV and high-β up to the final 2 GeV. The normal conducting linac and spoke cavities will be fed by a 352.21 MHz radio-frequency (RF), and the rest of the cavities by 704.42 MHz. Finally a High Energy Beam Transport (HEBT) will drive the beam to the target. Figure 1.2 shows a diagram of the accelerator.

- A five-tonne helium-cooled target wheel, where the spallation process takes place, converts the proton beam into slow neutron beams. Due to the low efficiency of the process, a great flux is needed for the experiments, which drives the requirements for the accelerated proton

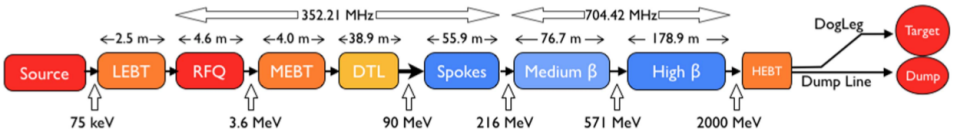

Figure 1.2: ESS linac layout. Source: ESS.

beam and its high power. Most of the 5 MW beam power will be dissipated as heat in the target, which will be located inside a 6000 tons shielding monolith [11].

The neutrons produced by the spallation process need to be moderated to extract the useful slow neutrons, which will be guided to 22 neutron instruments, each of them with a specific purpose and characteristics. The data generated by the instruments will be sent across the Öresund strait to the DMSC at the University of Copenhagen, in the Nørre campus.

ESS is a collaboration facility where the member states support the ESS project mainly through in-kind projects by providing systems which are engineered, designed and built in the members' homeland, and that are later shipped and installed locally at ESS. Meanwhile the host members of Sweden and Denmark provide most of the cash needed.

1.2 Controls and timing at ESS

The ESS Integrated Control System (ICS) is used in all parts of ESS: accelerator, target, instruments and conventional facilities. The software toolbox that was decided to be used is the *Experimental Physics and Industrial Control System* (EPICS) [12]. EPICS is an open-source environment developed by a collaborative community for developing and implementing real-time distributed control systems for big research centres and industry. It is widely used in particle accelerators, large telescopes and other experiments. It was developed to implement systems with a big number of computers linked by network, and it provides monitoring, interactivity, data archiving and control among other scientific applications. Communication is done following a distributed client-server model, where all nodes can act as client or server depending on the requirements. EPICS uses a special protocol called *Channel Access* (CA), that all nodes in the system use; the nodes are called *Input/Output Controllers* (IOCs). The new version 4 of EPICS extends its

functionality and adds better features and support.

ESS will use hardware based on its applications split in three layers [11]: slow industrial automation based on Programmable Logic Controllers (PLCs) for slow,reliable input/output (I/O); distributed I/O based on the EtherCAT standard [13] with real-time capabilities for applications with moderate data rates; and fast applications. The fast applications need real-time processing in the megahertz level, with data rates of hundreds of MBytes per second and early processing. For this applications the μTCA or MicroTCA (MTCA.4) [14] standard was chosen. One of the systems that are critical for the fast applications at ESS, and every accelerator or research centre in general, is the timing system. Its main purpose is to provide synchronization of several systems accross the facility the facility, which includes triggering devices at the specific time needed for the correct operation of the machine (usually with a configurable delay from a common source), providing synchronized clocks for data sampling and distributing other information reliably [15]. All of this must be achievable with tight requirements on drift and jitters, usually in the range of few ns, or even sub-ns for the clocks. Due to these requirements on stability and reliability the timing signals are usually implemented in hardware, while being controlled, managed and read from the control system.

The timing system is not only needed to synchronize the operation of all the different systems but also to set a common time-reference for the entire facility and guarantee that all acquired data and control signals can be correlated with coherent timestamps. The most common way to achieve this is by having a master that sends its local time and other related information across the facility through a dedicated timing network to a series of timing slaves or receivers that synchronize their internal time to the received one. There are two main timing system paradigms [16]:

- Time-based timing systems: in a time-based timing system the master, once the slaves have locked their internal time to that of the master, broadcasts messages with instructions expected to be performed at specific times. Then the slaves use their local time, derived from the master's, to know when to generate the requested triggers and signals.
- Event-based timing systems: in an event-based timing system the master is the one that waits until the requested time to broadcast events to the slaves, which react immediately as they are configured to. In this paradigm, an *event* is just an enumerated pulse signal that triggers

actions in the slaves. ESS uses an event-based timing system whose implementation will be discussed in this work.

Usually timing systems combine both paradigms, where each of them has obvious advantages: an event-based system can very easily drive actuators, especially when the timing system needs to react immediately or with low latency to external conditions. On the other hand knowing or being able to know the expected actions and times well in advance can help lowering the network traffic when a lot of actions are expected in a short time. Having a common time-reference for all the nodes in the system is needed in both paradigms for timestamping acquired data. It is worth noting that time-based systems with short deadlines between the broadcast of messages and the requested time of reaction is for most effects indistinguishable from a event-based system [16].

1.3 About this thesis

In this work we describe the technical implementation of the ESS timing system, in order to guarantee a common time reference for the facility, use coherent timestamps and synchronise the operation of all parts of the facility. This includes the explanation of the timing system from its fundamental working pieces, the design of a structure which allows the timing system to interface all the ESS subsystems that require synchronisation and the integration of the timing system in the ESS control system. It also includes a discussion about the synchronisation of the ESS facility.

This thesis is divided as follows: the present Chapter 1 gives an introduction about the ESS facility, the control system, a general presentation of timing systems and the scope of the work. Chapter 2 is concerned about the challenge of synchronising large facilities such as ESS, the ESS timing system is introduced in detail and explains the strategy to deal with the synchronisation challenge. Chapter 3 presents new developments in the technical implementation of the timing system at ESS and its integration in the ESS control system, both at low level and at EPICS level, and provides a description of the synchronisation strategy at ESS. Chapter 4 presents a new tool to quickly deploy FPGA systems automatically built from a high level graphical user interface description, and an alternative way to implement the same high level graphical user interface description as microprograms in FPGA-based systems. Finally, Chapter 5 presents the conclusions of this

work and summarises the next steps in the project.

Chapter 2

The ESS timing system

2.1 Synchronization

The main problem that the ESS timing system has to solve is how to trigger and control systems and devices that need to act in a synchronized manner when the distance between them is of hundreds of meters. The problem is two-fold: first, one needs to guarantee that the signal sent by the master is received at the slaves without alterations; second, one needs to guarantee that all slaves receive the signal at the same time, which in practice means with the same delay from the master, or with different but known delays and are able to compensate accordingly.

The simplest and easiest way of synchronising all the nodes of the timing system is by sharing a clock signal. In this context a clock signal is a repetitive sine or square wave (depending on if the system is analog or digital). Let us consider a simple sine wave signal[1]. One can write the periodic signal as a function of time (t) as

$$y(t) = A\sin(2\pi ft + \varphi(t)) = A\sin(\omega t + \varphi(t)) \tag{2.1}$$

where A is the amplitude of the signal[2], f is the frequency, ω the angular frequency and $\varphi(t)$ the phase offset. Equation 2.1 can be rewritten as

$$y(t) = A\sin(\omega(t + \frac{\varphi(t)}{\omega})) \tag{2.2}$$

[1] This analysis can be extended to square signals, since they can be reconstructed as a sum of sine wave signals, even being a finite sum in systems with limited bandwidth.

[2] Although the amplitude may have a dependency with time, let us ignore this dependency as it does not have any effect on synchronization.

where the $\frac{\varphi(t)}{\omega}$ represents the deviation between a perfect and an imperfect periodic signal. The phase offset $\varphi(t)$ is a random noise signal whose RMS[3] value can give one an idea of the quality of the clock [16].

The instantaneous phase $\phi(t)$ of the periodic signal can be expressed as

$$\phi(t) = 2\pi f t + \varphi(t) \tag{2.3}$$

Even for perfect periodic signals, where $\varphi(t) = \varphi_0$ is just a constant initial phase offset, two periodic signals with the same initial phase offset $\varphi_{1\_0} = \varphi_{2\_0}$ will have different phase relation over time:

$$\Delta\phi(t) = \phi_1(t) - \phi_2(t) = (2\pi f_1 t + \varphi_0) - (2\pi f_2 t + \varphi_0) = 2\pi t(f_1 - f_2) \tag{2.4}$$

This means that the only way to synchronise nodes is by sharing exactly the same frequency, as even a small difference will make them drift away from each other. For this reason the master of the timing system sends its clock to all the slaves that receive and lock to it, since it is the only way of guaranteeing that all nodes share the same frequency.
If at least two periodic signals have the same frequency it is said that they are syntonized.

In the case of periodic signals, once the nodes are syntonized and assuming that $\varphi(t)$ is a noise signal centred around the initial phase offset φ_0, and that two signals have different initial phase offsets φ_{1_0} and φ_{2_0} then

$$\Delta\phi(t) = \phi_1(t) - \phi_2(t) = (2\pi f t + \varphi_{1\_0}) - (2\pi f t + \varphi_{2\_0}) = \varphi_{1\_0} - \varphi_{2\_0} \tag{2.5}$$

the only thing left that one has to guarantee to have the nodes synchronized is that they have the same phase offset $\varphi_{1\_0} = \varphi_{2\_0}$. In this case

$$\Delta\phi(t) = \phi_1 - \phi_2 = 0 \tag{2.6}$$

This will guarantee that for example the rising edge of triggers will happen at exactly the same time across the facility.

2.1.1 Phase locked loops

Phase locked loops (PLLs) are used to guarantee that all nodes have exactly the same frequency (they are syntonized). They also clean the jitter

[3]Root mean square.

of clocks [16], which is the deviation of a signal from being perfectly periodical, showing up as variations in the phase of the signal. PLLs are basically control systems with negative feedback that generate an output frequency with a phase that has a certain relation to that of an input or reference frequency. They have four basic elements: a phase detector, a low-pass filter, a voltage-controlled oscillator and a feedback path. The phase detector (PD) compares the reference frequency with the output frequency, and creates a signal whose voltage is proportional to the difference of the phase of the two frequencies. The voltage signal is passed to the voltage-controlled oscillator (VCO) through the low-pass filter. The VCO is adjusted using the voltage signal to generate the output frequency. Finally the negative feedback is used to send the signal back to the PD so that the PLL can match the phases until it stabilizes. At that moment it is said that the PLL is locked. If the output phase drifts, the PLL, thanks to the negative feedback loop, will change the voltage signal in such way that the VCO drives it phase in the opposite direction to the drift, reducing the difference in phase between the reference frequency and the output. A diagram of a PLL is shown in Figure 2.1.

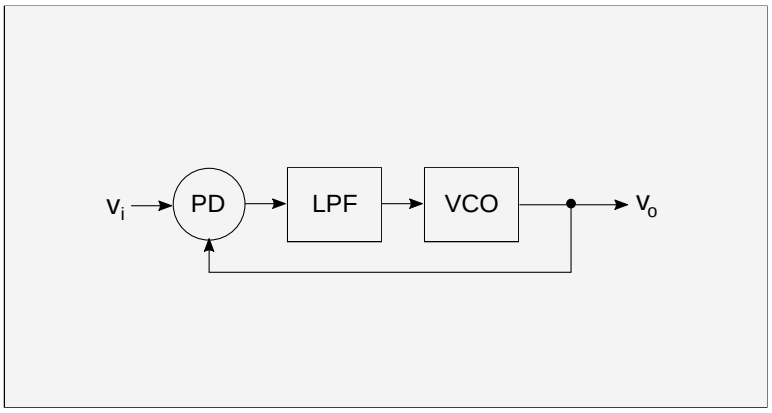

Figure 2.1: Phase locked loop.

In most timing systems the master does not send a periodic clock, but rather a data stream without an accompanying clock. Assuming that the data stream has enough transitions (for example by using some specific kind of encoding, such as 8b/10b, explained in Section 2.4.2) a PLL can use an internal clock, for example from an internal fractional synthesizer, to generate a clock which is phase-locked to the transitions of the data stream. This

process is called *clock recovery*.

2.1.2 Transmission delays

To achieve complete synchronisation one needs to guarantee that all the slaves in the timing system use the same origin of time, i.e. a specific instant in time that acts as a time reference has been defined and simultaneously distributed throughout the network, and is used by all slaves in the system. Then all nodes can correlate any other instant in time, even in different places, by counting cycles of the clock signal. Since it is usually not physically possible to distribute this reference instant to all slaves exactly at the same time, for example because of different link lengths, this is usually achieved by defining this specific instant in the master, broadcast it to the slaves, and somehow calculate or inform the slaves about the exact delay that it has taken for the signal to travel from the master to each of the slaves. Then the slaves can use this information to send triggers at precise times and calculate timestamps.

In Figure 2.2 a possible way of calculating the delay is shown, using a two-way scheme, and assuming both nodes are capable of sending information to the other and that they can timestamp the messages. One of the nodes sends a specific signal to the second node at a local time t_0, and the second node receives it at its local time (which may be different than the local time of the first node) t'_0. Then this second node replies with another signal at t'_1, which is received at t_1 by the first node. If then they share with each other these timestamps t_0, t'_0, t_1 and t'_1, they can calculate the delay in the link:

$$\Delta t = \frac{(t_1 - t_0) - (t'_1 - t'_0)}{2} \tag{2.7}$$

After knowing this delay it is usually the slaves that change their local time to align with that of the master. In general this calculation should be done independently for each slave, and depending on the topology of the network, for example in a star topology where there are one or more levels of cascaded fan-outs between the master and slaves, it should also be done for each level of the network. Also this way of calculating the delay of the transmission is only valid if the link is perfectly symmetric from the master to the slave as from the slave to the master.

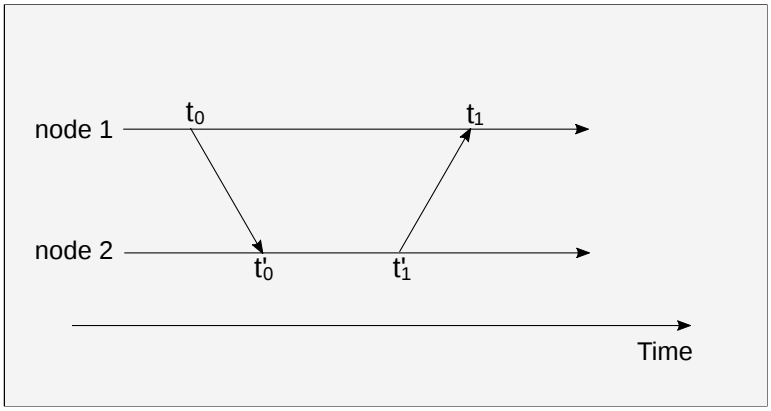

Figure 2.2: Calculating the delay of the transmission between two nodes.

2.2 Synchronization technologies

There are a number of synchronization technologies in use currently in the world, and most of them do not have accelerators or other research facilities as their main user, but it is actually the telecommunication industry that makes the most intensive use of them. In the following subsections some of these technologies are explained and compared, paying attention to the accuracy (which represents how close a slave can lock its frequency to that of the master) and precision (the jitter of the slaves measured from the average deviation that is defined by the accuracy) [16].

2.2.1 NTP

The Network Time Protocol (NTP) is used to synchronize the time of a system to another reference time source, usually a server or satellite receiver, over the network. Usually a series of servers is used to synchronize to a primary server that holds the Coordinated Universal Time (UTC) via a Global Positioning System (GPS) receiver. Several servers and different network paths are used at the same time to achieve better accuracy and reliability [17]. The accuracy of NTP is in the range of milliseconds due mainly to two facts: the first one is that the timestamping of the packets is performed in the software layer, suffering from the scheduling latency of the operating system. The second is that the propagation of the packets is performed through a network with variable latency due to the different paths and the non-deterministic latencies of routers and switches. This second factor can

be partially mitigated by using certain algorithms to calculate and compare the latency of different network paths, helping to achieve better accuracy and reliability.

2.2.2 PTP

The Precision Time Protocol (PTP) [18] was developed with the goal of providing better accuracies than those attainable with NTP while being based on the same idea, synchronizing times throughout a computer network. Although it may make use of software timestamping, to achieve the best accuracies hardware timestamping is needed. This is implemented by using dedicated PTP network cards for the grandmaster, switches (boundary clocks) and slaves. The PTP grandmaster is the origin of the time in the network, and is usually disciplined by GPS or similar. As the switches and slaves drift from the grandmaster clock due to misalignments in the internal clock frequencies, continuous compensation is needed in them. In this way PTP can attain accuracies in the microseconds range.

2.2.3 Other current technologies

Current research facilities need synchronization even better than in the range of microseconds, as usually the minimum expected accuracy is one beam bunch or bucket of the RF used in them, which is typically around 500 MHz for modern light sources and electron storage rings [19]. In this case the main challenge that the timing systems need to address is the drift of the slaves' oscillators compared to the master's. The main solution for this issue is to recover the master's clock signal from the data stream at the slaves. In this implementation it is the master's clock signal that is used as the encoding signal of the bitstream. The slaves lock to the data stream (by using a PLL) and extract both the messages and the encoding signal, which they use internally for their local counters. In this case, and since the slaves are using the same clock for the timestamps as the master, it is possible to reach accuracies in the range of nanoseconds [16]. Except for thermal drifts, which are slow and may be compensated by methods such as the one shown in Figure 2.2, it is guaranteed that there is no frequency offset between the master and the slaves.

Examples of this kind of timing systems are the MRF timing system,

used at ESS and which will be later described in Section 2.4, and White Rabbit [20]. White Rabbit is a collaborative project among several scientific facilities, universities and companies, led by CERN and the GSI Helmholtz Centre for Heavy Ion Research (GSI). Its goal is to develop a timing and synchronization network, completely open (both hardware and software), based on Ethernet, with thousands of nodes with distances of around 10 km between them, and with sub-nanosecond accuracy. It is based on PTP together with hardware timestamping and Synchronous Ethernet (SyncE) [21] to transfer the clock signals over the network.

2.3 Definitions

In this section a series of concepts with a very specific meaning in the context of the ESS timing system and that will be used in the rest of this thesis are defined.

2.3.1 Beam cycle

In the context of the timing system, a beam cycle is one period of the 14 Hz cycle of the accelerator operation. Each beam cycle is 71 ms long and includes one 2.86 ms window where extraction and acceleration of protons happens. The beginning and end of every cycle are defined in such way that all events and timing transmissions needed for one specific beam pulse are included in the same cycle.
Purpose: extraction and acceleration of the protons that trigger the spallation process in the target.

2.3.2 Beam pulse

The beam pulse is the actual group of protons that travel through the accelerator. In normal operation it can also refer to the 2.86 ms window where protons are extracted and accelerated.
Purpose: trigger the spallation process in the target.

2.3.3 Event

An event is an enumerated signal sent from the event generator (timing master) to the event receivers (timing slaves). The event receivers receive the event and act based on it, usually by triggering an output with configurable delay and width, although events can also set or reset outputs, cause hardware interrupts or trigger some processing. Special events are used to broadcast the timestamp (real world time), synchronize prescalers or perform special functions.
Purpose: broadcast of an enumerated signal throughout the ESS facility with a deterministic latency. They can be used to generate triggers.

2.3.4 Trigger

Triggers are signals created by event receivers (timing slaves) to synchronize external equipment or cause the start of processing. Usually they have an interface with the equipment through cables or the μTCA backplane.
Purpose: trigger external equipment.

2.3.5 Data items

Each of the data items corresponds to one parameter of the proton beam, such as beam length, beam mode and so on. All the data items are sent as part of one big data buffer of 2 KiB, which is divided in segments.
Purpose: transmission of beam-related information.

2.3.6 Event frequency

The frequency at which the events are sent. In the case of ESS, it is 88.0525 MHz, divided from the frequency of the master radio-frequency oscillator and thus phase-locked to it. The data buffer is sent at a rate of one 8-bit byte every other tick of the event clock (which runs at the event frequency). The clocks are sampled at half of the event frequency.
Purpose: sending events, data and clocks in a synchronous way.

2.3.7 Sequence

The sequence is a list of up to 2047 event-timestamp pairs (in the context of a sequence, the timestamp is the delay from the start of the sequence, do not confuse with the regular timestamp). Once triggered, it sends the events in the list at the time stated by the corresponding timestamp. The sequence is used to define the relation between different events.
Purpose: sending a series of pre-defined events at pre-defined times.

2.3.8 Supercycle

The supercycle is a pre-defined series of sequences that run in automatic mode. It is to be used for the commissioning of the accelerator, to slowly increase parameters such as the beam pulse length pulse-by-pulse. The supercycle is also an operating tool under development, with a set of applications to create and run a set of cycles.
Purpose: slowly change the beam parameters in a pre-defined and automatic way.

2.3.9 Timestamp

In the context of the timing system, a timestamp is a record of the exact moment when an action takes place in relation to a centralised "real world" time, to allow correlation of data from different parts of the facility. Do not confuse with the sequence parameter also called timestamp.
Purpose: precise time correlation of actions.

2.3.10 Clock

In the context of the timing system, a clock is a periodic signal.
Purpose: synchronisation of systems with a periodic signal.

2.3.11 Real world time

The real world time is the normal time (year-month-day-hour-minute-second) as presented by a NTP server or a GPS receiver.
Purpose: common reference for timestamps.

2.4 The ESS timing system

ESS uses the timing system developed by Micro-Research Finland (MRF) [22]. This is an event-based timing system that provides a complete tree-network distribution of timing signals with just three components: an Event Generator (EVG) which acts as timing master, Event Receivers (EVRs) which act as timing slaves, and fan-out modules. It generates and distributes synchronous clocks, sequences of events and trigger signals synchronous to an external source, usually the master clock for the RF signal that also feeds the accelerating and control devices in the accelerators, storage rings and beamlines. This external RF signal is needed to guarantee that all clocks and signals are phase-locked to the machine operation. The MRF timing system can synchronize machines with sensitive systems and electronics to the AC current mains (50 or 60 Hz) [15]. This timing system also provides timestamping functionality for defining a common time reference and identifying in time the actions performed and the data acquired in the facility. It was developed by a single company but in close relation with the experimental physics community, with its requirements driving the design. Part of the design is open [23] and the protocol is well known. It has been used for many years in several facilities such as the Linac Coherent Light Source (LCLS) FEL [24] at SLAC in California, USA, the Swiss Light Source (SLS) [25] at the Paul Scherrer Institut in Switzerland and Diamond Light Source [26] in the UK, both of them synchrotron light sources, or the Facility for Rare Isotope Beams (FRIB) [27] in Michigan, USA. In all of them it has shown very good reliability, making it one of the reasons why it was chosen at ESS. Another reason is that the supplier keeps developing and enhancing all the timing system products, decreasing the risk of product obsolescence [11].

The EVG defines the time to which the EVRs lock to, and sends the events, clocks and other signals that synchronize the facility. It is highly configurable and self-contained, without needing external devices such as counters, etc.

The EVRs lock to the link from the EVG, decode the data stream and act based on it, basically by producing both hardware and software triggers from the events and timestamping the actions. This makes the system very flexible.

The ESS timing system has five main functions:

- Event distribution throughout the facility, which are mainly used to create triggers.
- Generation and distribution of synchronous clock signals.
- Definition of a common time base and timestamping throughout the facility.
- Transmission of fast and synchronous beam-related data.
- Delay compensation for stability against long term thermal drifts.

2.4.1 Topology

The most basic set-up of the timing system consists of an EVG connected, through a fan-out layer, to a number of EVRs. The fan-out layer can have any number of levels of cascaded fan-out modules and it is not necessary for it to be balanced. It is possible to mix EVRs and fan-out modules in any level of the topology. Devices are aware of their position in the tree structure and get a unique identifier according to it. For synchronization with the machine, mainly the RF devices, the EVG has an input for the RF signal, which is used to derive the event clock from, as explained later. Another EVG input is used to keep the timestamps synchronized to the world via a GPS receiver which shares a 1 pulse-per-second (PPS) signal. Figure 2.3 shows a simplified diagram of the timing topology.

Delay compensation

An active delay compensation mechanism measures the propagation delay between every two nodes directly connected to each other in any level of the topology, so that each node can calculate the total delay from the EVG. This is done by adding the delay of its level to the total delay of the previous levels, which is transmitted downstream. On the lowest level of each branch each EVR knows the delay from the EVG down to its own level, and uses that information to adjust the depth of an internal event FIFO[4] buffer to guarantee that the events reach the processing core of each EVR after a programmed target delay. In general this target delay should be the same for all the EVRs in the facility.

[4] First-In-First-Out.

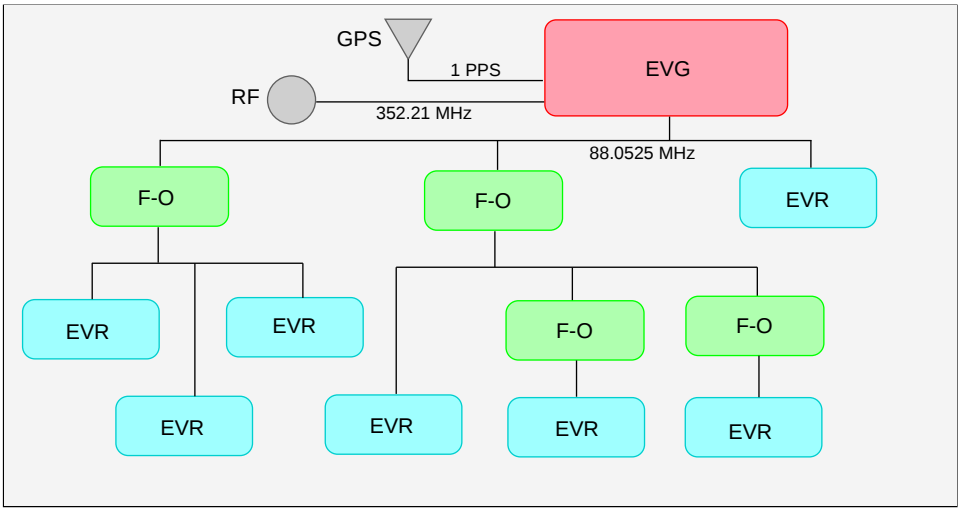

Figure 2.3: Topology example of the ESS timing system.

The event clock

The timing system needs to be synchronized to the machine, basically to the RF signal that feeds the accelerating devices. One option would be to use two different clocks, one for the RF and another one for the timing system, both with the same frequency, namely 352.21 MHz, used for the ESS accelerator. This frequency is fixed and cannot be changed due to some parts of the accelerator being already built with a strong physical dependency to this frequency. If the clocks are off by just 0.001 parts-per-million, in approximately 3 seconds both clocks would already drift from each other by one cycle. This is obviously unacceptable, so the only choice is to have only one master, the RF oscillator, and feed that frequency to the timing system, for it to be used to derive the timing event clock from.

The timing devices, EVG and EVRs, need to create and react to events at the frequency of the event clock, and some complex operations are needed to be performed on some of those events. For this reason the timing system provided by MRF has been developed to use event clocks from 50 MHz to 142.5 MHz, guaranteeing that it will perform as expected in that range of frequencies. The EVG can accept external RF frequencies from 50 MHz to 1600 MHz, and internal dividers will divide that input frequency by an integer number configurable by the user (from 1 to 32) so that the event frequency is in the supported range. As mentioned before ESS will use a

RF frequency of 352.21 MHz, which will be divided by 4 to obtain an event frequency of 88.0525 MHz. Events will be sent from the EVG to the EVRs at this event frequency, which are phase-locked to the RF frequency. The EVG can sample the RF frequencies with four phases at 0°, 90°, 180° and 270° to synchronize the event clock to.

As it will be explained in the next sections, the events are sent through the timing network coded in a digital signal, so that the original clock can be re-constructed at the EVRs. On top of the two clocks already mentioned, one with the RF frequency and the other with the event clock frequency, a third clocks is also generated, the bit clock for the transceivers in the timing network. The relation between this clock and the event clock is fixed at a factor of 20, as will be explained in the following sections. The relation between the three clocks is then as follows:

$$F_{bit}/20 = F_{event} = F_{RF}/N_{divider} \tag{2.8}$$

where F_{bit} is the frequency of the transceivers, and for the case of ESS, the event frequency F_{event} is 88.0525 MHz, the external RF frequency F_{RF} is 352.21 MHz and the EVG divider $N_{divider}$ is 4.

2.4.2 The data stream

The data stream is distributed through the facility using an optical fibre network, specific for timing. Since the delay compensation is based on the calculation of the delay on the fibre between two adjacent nodes that exchange messages assuming that the link is symmetric, according to Equation 2.7, it is of uttermost importance that the delay upstream and downstream is the same. Although there are available several different ways of dealing with this, the easiest way is using duplex fibre and the same light frequency on both strands of fibre. This ensures that the travel path of both ways is the same within some tolerance, and the same wavelength and diffraction index also ensures that the optical path is the same. In this way a symmetrical link is guaranteed.

Each cycle of the event clock a frame consisting of two 8-bit words are sent through the timing distribution network, from the EVG down to all the EVRs. The first word represents an integer that symbolizes an event. There are 256 different event codes, each of them with a different meaning

set by the timing developer, although some events are already in use internally by the timing system to perform some actions, such as transmitting the timestamps. Only one event may be transmitted on each cycle. Event number zero is the null event, and is sent when no other event is scheduled. The second word is shared by the data buffer and the distributed bus. The data buffer is just a 2 KiB buffer of memory which is copied from the EVG to the EVRs, byte by byte, and is explained in more detail in Section 2.4.2. The distributed bus consists of eight independent clocks which are sampled at a frequency derived from the event clock, and one sample of each clock is combined to form a word of the distributed bus sent over the data stream. More about the distributed bus is explained in Section 2.4.2.

There are two ways of configuring the data stream. In both of them the first word is always an event clock, while the second word is different depending on the configuration. In the first configuration the second word is always part of the distributed bus, so the eight clocks are sampled at the event clock frequency, 88.0525 MHz for ESS, so they can transmit frequencies of up to approximately 44 MHz. In the second configuration the second word of the data stream is shared between the distributed bus and the data buffer, so that a word of each is sent in alternating event clock cycles. In this case the distributed bus clocks can achieve a maximum frequency of approximately 22 MHz and the full data buffer is sent in approximately 46.5 microseconds.
ESS will always run with the second configuration, using the data buffer and the distributed bus. The frame structure is shown in Figure 2.4.

The 8b/10b encoding protocol

The frames that form the data stream are sent continuously downstream creating a synchronization clock which allows the local oscillator of each EVR to recover the event clock sent from the EVG, which is highly related with the RF signal as explained in Section 2.4.1. In order for the event clock to be retrieved from the link frames, the data stream needs to meet some requirements, mainly on the number and frequency of the transitions of the digital signal. To achieve this the data frames are encoded using the 8b/10b protocol [28], which on top of guaranteeing that the maximum run length is five for clock recovery it also achieves DC-balance by ensuring that in any long enough string the difference between 0's and 1's is no more than two. In order to do this the protocol transforms the 8-bit words into specific 10-

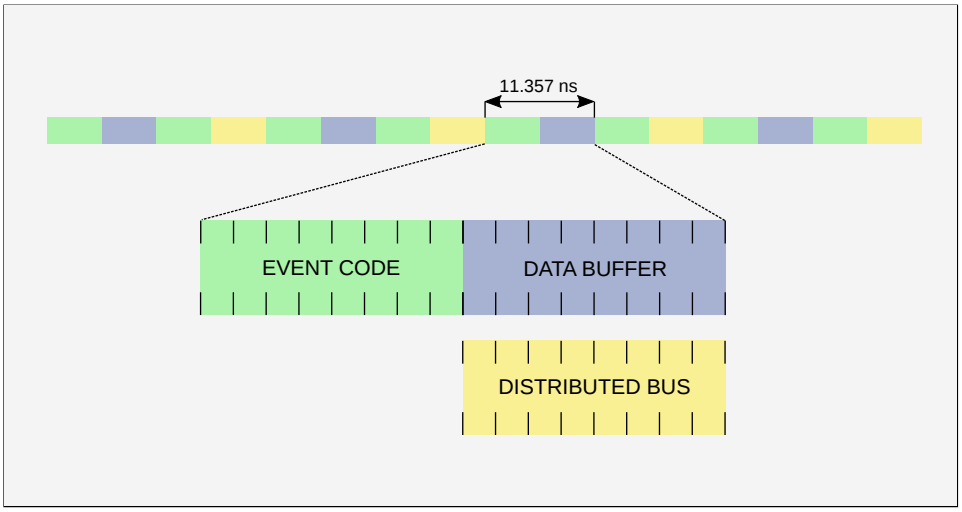

Figure 2.4: Frame structure of the data stream.

bit words, which comply with the previously explained characteristics and allow the decoding of the original words. Every now and then a null event is replaced by a special 8b/10b synchronization character to synchronise the boundaries of the data words in the serial data stream. Since each frame consists of two 8-bit words and the event clock frequency at ESS is 88.0525 MHz, it can be calculated that the timing distribution network works at a frequency of approximately 1.76 Gbps.

Timing events

Events are instantaneous pulses that are sent from the EVG to the EVRs, intended to be used to generate physical triggers in the EVR outputs and start software processes. Events can be considered independent digital pulses, but only one can be sent each cycle of the event clock. Event 0 is the null event that is sent when no other event is queued.

There are several sub-systems in the EVG capable of sending events:

- The event sequencer: it is a list of 2048 entries, each entry being formed by an event and its related timestamp. In the event sequencer context, the timestamp is the delay from the moment when the sequencer was triggered, and is expressed as an integer number of cycles of the event

clock. All the entries need to be sorted in ascending timestamping order, since its working mechanism is as follows: when the sequencer is triggered, the first entry in the list is fetched, and a counter counting event cycles starts running. When the counter reaches the value of the timestamp, the related event is sent, and the second entry in the list is fetched. Again, when the counter reaches the value of this second timestamp, the second event is sent, and the following entry is fetched, and so on. The sequencer keeps running until the end of the list, which is marked by the event 127 (0x7f), that is a special event and is not sent. The sequencer is used to play back sequences of events in a specific order and with well defined times between them. The list is stored in RAMs[5] of which the EVG has two able to run in parallel, although the intended use is to write a new list in one of them while the sequence in the other RAM is running.

- Multiplexed counters: they are independent 32-bit counters that generate clock signals with frequencies derived from the event clock. They do not send events directly, but are used to trigger the sub-systems that do send events. Each multiplexed counter is configured with an integer value, and when that value is reached, the multiplexed counter triggers and resets, starting counting again. The counters are updated with the event frequency, so the range of frequencies that they generate go from half the event clock to $\frac{eventclock}{2^{32}-1}$ (in the case of ESS, this means frequencies from approximately 44 MHz to approximately 0.02 Hz). Multiplexed counters can be used to send events, trigger the sequencers or drive the clocks of the distributed bus. The EVG has eight multiplexed counters with 50 % duty cycle (for even divisors, odd divisors will be slightly off).

- Trigger events: they send a unique event every time time that they are triggered. The event to be sent is configurable by the user. The possible triggers are the EVG's inputs and the multiplexed counters. The EVG has eight completely independent trigger events. If the trigger events are driven by the inputs, the signals are synchronized to the event clock.

- Software events: they are used to send a unique event when that event code is written to one of the EVG's registers by the user or operator. Because of the way that they are triggered, they are not deterministic

[5] Random Access Memory.

as the rest of event sources, and are meant to be used to perform tests or trigger non-timing-sensitive actions.

On top of these sub-systems, the EVG can automatically send events that drive the timestamping sub-system, as is explained in Section 2.4.3. The EVG has a priority encoder to resolve which event to send when two events are scheduled in the same cycle. The lowest priority event will be sent in the next empty cycle.

The EVRs are also capable of sending events upstream from a number of sources, mainly their inputs, in parallel to the normal downstream link. This capability is supposed to be used as a fast communication mechanism to react to some physical occurrence downstream in the accelerator by stopping the EVG normal behaviour, and hopefully prevent damages to the machine or people. The protocol used upstream is exactly the same as the one used downstream: there are 256 different event codes, with only one event being sent every cycle of the event clock. In this case the fan-out modules work as concentrators with an obvious risk of acting as bottle-necks, so this feature should be used cautiously. When the upstream events reach the EVG, the EVG stops the transmission of regular events and the upstream events are replicated broadcast downstream to all EVRs. The EVRs react to these events by sending triggers to some devices that hold circular buffers that keep the running state of the machine. The intention of these *post mortem* system devices is to freeze the circular buffers when they receive the triggers from the EVRs, saving the exact state of the machine in the moments prior to the fault, so that it can be studied and fixed.

EVRs can also create local events from inputs, with the intention of timestamping the signals connected to those inputs, and also triggering a sequencer when running in standalone mode, as explained in Section 3.1.

Data buffer

The data buffer allows sending beam-related information from the EVG to the EVRs in a fast and reliable manner. It is a block of 2 KiB of dual-ported buffer memory that is written to the EVG, and, when triggered, is copied to the EVRs. The EVRs raise a flag when the data buffer transmission is running, and another one when the transmission has finished, by using especial 8b/10b codes. The transmission is done byte by byte each

other frame of the data stream, and is triggered by software, so although the transmission itself is time-deterministic, its triggering is not, so the data buffer should not be used for triggering, just for transmission of information.

The data buffer has no internal structure, so it is up to the users to organize it internally as they want, and also to make sure that the EVG and the EVRs expect the same information and that it is expressed in the same way, although the implemented protocol performs error detection with a simple checksum. ESS has implemented a protocol for the data buffer explained in Section 3.4.3. The last bytes of the data buffer are reserved by the timing system to exchange information between nodes about the topology and the delay compensation.

At ESS the data buffer will be sent during one beam cycle with the beam related information concerning the following beam cycle. The intention of this is that the information is available to the consuming systems with enough time to get ready and, if for some reason they cannot, inform the timing system or MPS[6] to abort the beam pulse. For this reason the data buffer should be transmitted as soon as possible, ideally right after the current beam pulse has finished. This would give the consuming around 60 ms to get ready for the following beam cycle.

Clocks (distributed bus)

The distributed bus consists of eight simultaneous clocks used to transmit eight independent signals sampled at the event clock frequency or half of it, depending on the data stream mode. The clocks can send frequencies sampled from the EVG's external inputs or the multiplexed counters. If more than one source is selected, all of them are logically OR-ed. One of the clocks can also be configured to reset the counter that updates the timestamps in the EVRs. In the EVRs the clocks of the distributed bus can be routed to the outputs.

2.4.3 Timestamping

Timestamping is the process of attaching a precisely time-tagged label to each piece of data saved in the facility. In the ESS case this is performed

[6]Machine Protection System.

by the timing system, allowing the saved data to be synchronized and correlated to the actions happening in the facility, such as the timing events. The timing system only timestamps the events, so everything that needs to be timestamped should have a related event. The actions and processes driving the facility are already triggered by events, so no further action is needed. For other signals that need to be timestamped, they should be fed to an EVR via its inputs. The EVR will synchronize the signal to the event clock, use it to create an event (either edge-sensitive or level-sensitive) and timestamp that event. In some cases the event may be sent upstream to the EVG, as configured by the user.

Timestamps have two parts: a "second" part and a counter (sometimes called "nanosecond") part. The "second" part is just a 32-bit integer counting the number of seconds in the UNIX epoch (the number of seconds since 1st January 1970 UTC, ignoring leap seconds). The "second" part is generated in the EVG and transmitted to the EVRs once per second. To generate the first timestamp when the EVG starts running, at ESS the EPICS layer that is used to configure the EVG uses the system clock of the device controlling the EVG, which should be synchronized to an external highly reliable source such as an NTP server or GPS server. After that the EVG uses an externally supplied 1 PPS signal, for example a rubidium clock disciplined by GPS, to increment the "second" part by one and send the new timestamp to the EVRs triggered by the 1 PPS signal. This happens at the rising edge of the input. The EVG also sends the 1 PPS signal to the EVRs using a dedicated, reserved event, driven by one of the trigger events. The EVG sends the new timestamp to the EVRs as a string of 0's and 1's using two dedicated, reserved events. The timestamp sent represents the exact second that applies in the real world the next time that the 1 PPS event is sent to the EVRs. If desired the EVG can resynchronize to the system clock, although this is not usually done.

The counter part of the timestamp is local to each EVR. It is implemented as a free running 32-bit counter, updated with the event clock and increased by one each cycle, that is reset with the 1 PPS signal sent from the EVG. When the 1 PPS is received by the EVR, the new "second" part of the timestamp is used together with the recently reset counter part. When an event reaches the EVR, from any event source, the EVR copies both parts of the timestamp and assigns it to the event. On top of the "seconds" part already explained, the counter part is used to calculate the fractional part of the timestamp. Since it counted the number of cycles of the event clock since

the start of the current second, the fractional part of the timestamp can be calculated by multiplying the value of the counter part by the period of the event clock (since the counter part of the timestamp stores an integer with a time value that depends on the event clock frequency, the "nanosecond" name that it sometimes receives is not really appropriate, although it does represent a number in the range of nanoseconds). In the ESS case the period of the event clock is approximately 11.357 ns, so this is the granularity of the timestamps. The multiplication is done in the EPICS layer.

EVRs can timestamp external signals as explained by creating events driven from the inputs. EVRs can also be used by the EPICS layer of the device that is used to control the EVR as NTP servers, to synchronize their system clocks to the timing system.

EVRs can not verify the timestamps without an external, trusted source, although some sanity checks are performed in the EPICS layer. Every second, the new "seconds" part is expected to be an increment of one from the previous second. If it is not the timestamp is declared as invalid, and the EVR waits for five consecutive, sensible timestamps to arrive from the EVG before setting the timestamp back to valid. It is also possible to detect when the 1 PPS signal is malfunctioning by expecting the signal in some "1 second + delta" time. If the signal does not arrive, the timing system raises an alarm and invalidates the timestamping.

2.4.4 Hardware

ESS will use three of the products developed by MRF: the mTCA-EVM-300, the mTCA-EVR-300(U) and the PCIe-EVR-300DC, which are described in this section.

The event master

The event master (EVM) is the physical implementation of an EVG and a fan-out module in the same card. ESS will use the same mTCA-EVM-300 card from MRF, shown in Figure 2.5, as EVG and fan-outs. The core of the EVM is a Kintex-7 model XC7K325T FPGA by Xilinx [29]. The EVM, when used as an EVG, is in charge of creating an event clock derived from an external RF source, and use that event clock to encode and

distribute events, a data buffer, a distributed bus and timestamping capabilities. On top of those it includes event-generation capabilities with eight trigger events, two event sequencers and eight multiplexed counters. It also includes a timestamp generator and the delay compensation mechanism. It includes a fractional synthesizer to create a number of frequencies to be used as event clock when an external RF source is not available, which is mainly used for testing purposes. A Micrel SY87739L Protocol Transparent Fractional-N synthesizer is the on-board chip, and has a reference clock of 24 MHz. In the front panel the EVM has eight SFP module connectors and two LEMO EPK.00.250.NTN connectors with an input impedance of 50 Ω. The first one used for feeding the RF frequency from the master oscillator, so that the timing system events are phase-locked to the RF and does not drift from it. The RF input accepts a square or sine signal with a maximum level of +10 dBm. The second LEMO connector, for general purpose and TTL level, will be used at ESS to feed the 1 PPS signal. It can accept 5 or 3.3 V TTL logic signals. Through the Rear Transition Module (RTM) connector the EVM includes more inputs, allowing the MPS to stop the generation of events that synchronize the facility when an important problem, that may damage the machine, is detected. Through the μTCA backplane more FPGA transceivers are routed to communicate with EVRs if installed in the same crate.

Although not used at ESS, the EVM can also be synchronized to the alternating current (AC) mains used for electrical power distribution. In other facilities this is needed for synchronizing the distribution of events to the frequency of the mains, since some devices will have different performance if triggered at different phases of the mains frequency. This is mainly a concern for the modulators that provide energy to the accelerating cavities, since different phases of the mains will cause different power to be supplied to the cavities.

When used as a fan-out the EVM will use the SFP port number eight for the upstream and the rest as downstream ports. In this case the EVM will decode the data stream from upstream and forward it downstream. The fan-out uses an independent delay compensation sub-system, so that it can correctly measure the delay for each port independently of the length of the fibres connected. Although the EVM acting as a fan-out can create events by itself, either from the inputs, triggers events, etc, and include them in the data stream, there is no intention of using this capability at ESS.

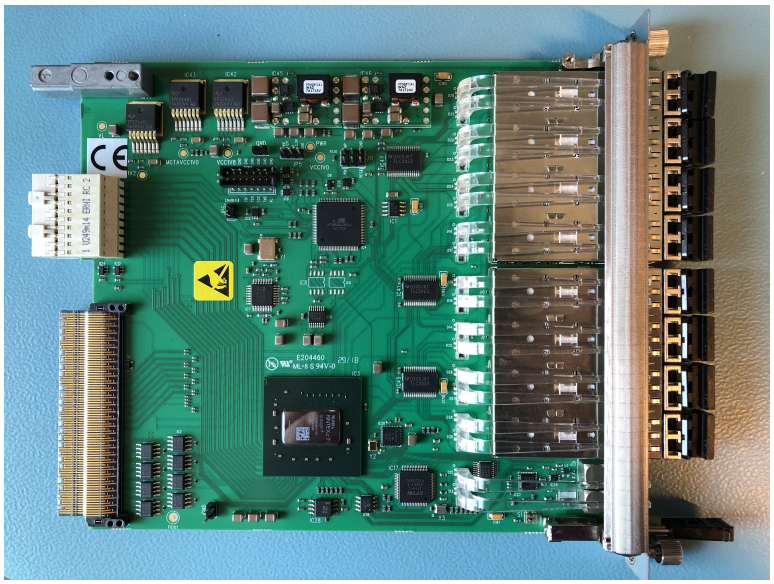

Figure 2.5: The mTCA-EVM-300 card.

The event receiver

The EVRs main uses are providing trigger signals to other devices from the events it receives from the EVG and timestamping those events as well as external signals. EVRs can be described as a series of sub-systems interfacing the data stream from the EVG to other external devices that need synchronization. These sub-systems are:

- Mapping RAM: it is a list of all the event codes and how the EVRs should react to each of those events. Each event can cause many independent actions, and also the same action can be caused by several different events. There are two basic groups of actions: pulse generator actions and special actions. The special actions are the ones related to timestamping, saving events to a FIFO that can be read by the EPICS layer, forwarding events upstream, acting as a timing heartbeat, logging events, blinking a LED, logging and freezing a circular event log buffer and resetting the prescalers. The pulse generator actions are setting the pulse generators to high level, setting the pulse generators to low level or triggering the pulse generators for a specific time after a specific delay.

- Pulse generators: also called pulsers, the pulse generators are driven

or triggered by the mapping RAM. When triggered, the pulse generators will react with their associated width, delay and polarity to drive a logical signal to high level (or low level, depending on the polarity) after the specified delay from the moment the pulse generator is triggered and for the selected time (width). The width and delay are specified as integer numbers, representing the number of cycles of the event clock. Pulse generators also have prescalers that divide the event clock by an integer number before being forwarded to the delay and width counters. Different EVR form factors have a different number of pulse generators, with variable prescaler, width and delay sizes. Some pulse generators do not have associated prescalers.

- Prescalers: they work in a similar way to the pulse generators prescalers but for special signals. Each of them divide the event clock by an integer number to drive a logical signal. All prescalers can be phase-synchronized by sending an event which is configured in the mapping RAM to reset the prescalers. If this event is local to the EVR only the local prescalers will be reset, but the event can also be sent from the EVG to act on the whole timing system. Each EVR can use different resetting events by writing different configurations to the mapping RAMs.

- Outputs: they connect each of the pulse generators, prescalers, distributed bus clocks to the physical outputs of the EVR. They can also be set to high, low and high impedance level. Only one source can be connected at a time to each output. There are four kinds of physical outputs:

  - Front panel outputs: these are the normal outputs located in the front panel of the different form factors. They are simple 3.3 V TTL level LEMO EPK.00.250.NTN connectors with an input impedance of 50 Ω.
  - Universal outputs: they are used to install modules with different kinds of output connectors to the EVRs. Each module has two physical outputs. The options are the 3.3 V and 5 V TTL LEMO EPK.00.250.NTN, LVPECL LEMO EPG.00.302.NLN, Avago HFBR-1528 Versalink optical transmitter and Avago HFBR-1414 optical transmitter.
  - Backplane and RTM outputs: these outputs are directly connected to the form factor's specific bus lines.

  - High speed pattern outputs: some special outputs allow sending high frequency signals derived from the event clock. EVRs use the recovered link stream clock signal to generate a clock that has the frequency of the event clock multiplied by 20. EVRs can use this clock to create four arbitrary patterns that are triggered with the "state" of the source signal (the pulse generators, etc); this "state" is the rising edge of the signal, the high level, the falling edge, and the low level.

- Inputs: used by the EVRs to create events that are timestamped, sent upstream or trigger some action locally, in the same way as the downstream events. They are realised by a LEMO EPK.00.250.NTN connector in the front panel and as a universal module with two of these same connectors. Events can be generated from the rising or falling edges of the input signal, or periodically when it is in high or low level.

- Timestamping mechanism: the EVRs decode the timestamping events included in the link stream from the EVG and keep the time internally. EVRs then use this time to timestamp events.

- Event FIFO buffer: all events marked in the mapping RAM to be included in this FIFO buffer are saved so that the EPICS layer can retrieve them. When the events arrive they are placed in the buffer alongside their exact time of arrival according to the local timestamp mechanism (both the second and the counter part). When the FIFO is not empty it raises an interrupt from the EPICS layer to retrieve the contents of the FIFO. If events are arriving at the EVR too quickly the FIFO will overflow, raise a "full FIFO" interrupt and loose further event occurrences. In the EPICS layer the retrieved events may trigger actions as well. If all of the actions triggered by one specific event are not completed before a re-occurrence of the same event it is also marked in the EPICS layer, although this is less serious than the full FIFO situation since other events and all hardware actions in the EVR are not affected.

- Data buffer: used by the EVRs to share with the system where they are installed some beam-related parameters sent from the EVG. When the EVRs receive the data buffer transmission complete signal they copy the content into an EPICS array. The EPICS layer is aware of

the internal organisation of the data buffer, segments and names each section, so that it is available to other processes in the system.

The EPICS layer can also use the internal time of the EVRs provided by the timestamping mechanism as a time source for the NTP daemon in Linux systems. This is achieved by writing the timestamp data to a shared memory segment from the EPICS layer and using a driver that reads the reference clock from a that memory segment. Since this method involves the EPICS layer, which runs in software, the provided time has a precision in the range of microseconds.

ESS will use EVRs in two form factors: the mTCA-EVR-300(U) and the PCIe-EVR-300DC, both provided by MRF and based on a Kintex-7 model XC7K70T FPGA by Xilinx. MRF also provides the configuration bitfiles with the firmware for the FPGAs, although a tool developed during this thesis and presented in Chapter 4 could be used to developed custom firmwares. The mTCA-EVR-300(U) comes in two flavours: the mTCA-EVR-300 and the mTCA-EVR-300U. Both of them including a SFP module for connecting to the timing distribution system, two TTL level inputs with LEMO EPK.00.250.NTN connectors, four TTL level outputs with the same connector, 32 RTM outputs and ten backplane lines. Of the ten backplane lines, eight of them are regular outputs and the other two are high speed pattern outputs with low jitter used mainly to share the event clock and other high frequency clocks. These backplane line outputs are the most used at ESS since they provide reliable and installation with less cables, since all the fast acquisition at ESS will be performed by other μTCA AMCs[7] and RTMs [30, 31]. On top of that the mTCA-EVR-300U has two slots for universal outputs while the mTCA-EVR-300 replaces these two slots by a micro-SCSI high density connector. The micro-SCSI connector is used to interface the IFB-300 board which has eight slots for more universal module outputs and inputs. Figure 2.6 shows a mTCA-EVR-300U. The PCIe-EVR-300DC, due to its small form factor, only has in its front panel a SFP module and the same micro-SCSI connector that the mTCA-EVR-300 uses for interfacing the IFB-300 board. Figure 2.7 shows a PCIe-EVR-300DC.

[7] Advanced Mezzanine Cards.

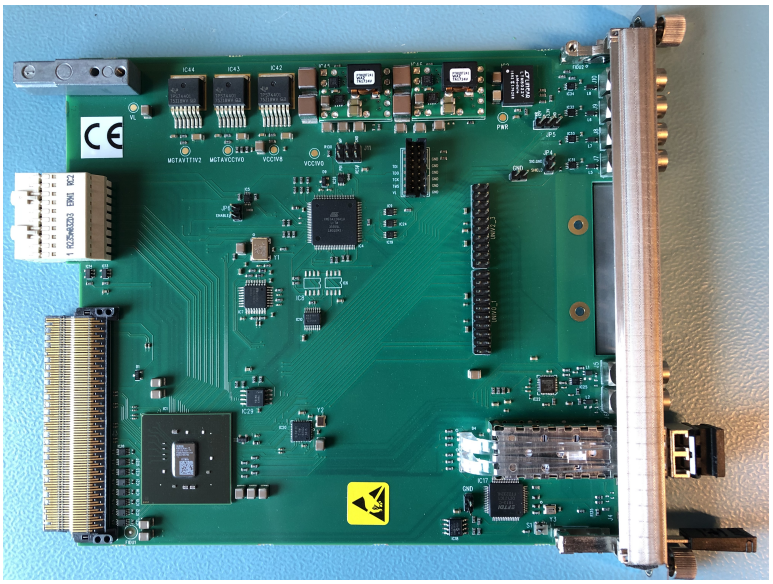

Figure 2.6: The mTCA-EVR-300U card.

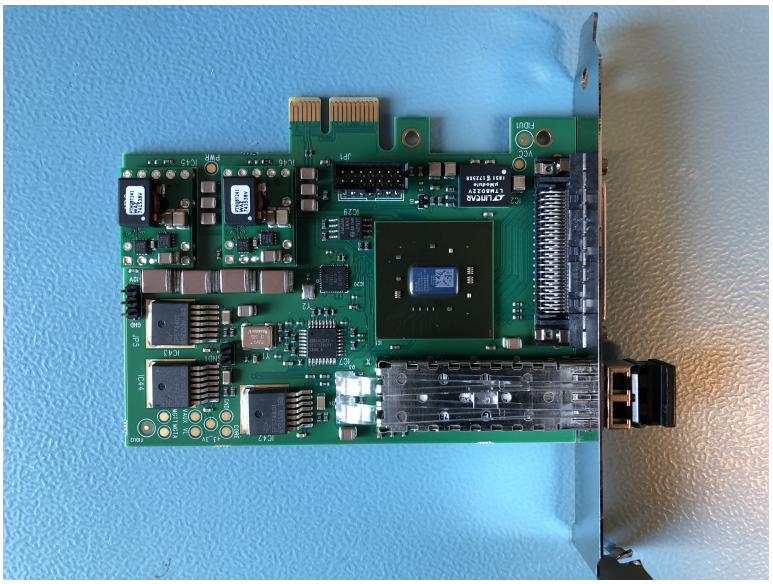

Figure 2.7: The PCIe-EVR-300DC card.

2.4.5 The integration of timing in the ESS control system

The timing system needs to be integrated into the ESS control system to allow its configuration and use. The first part of this integration is the

hardware layer. Most of the equipment used at ESS will be in the μTCA form factor, and so are the EVRs, so that the μTCA characteristics can be exploited. Each μTCA crate will have an AMC processor by Concurrent Technologies [32] that will act as crate master, including the PCI bus, and that will configure the EVRs. This is done by writing to and reading from the appropriated registers of the EVR via the backplane PCI bus. The EVG is also implemented in the μTCA form factor and will be controlled in exactly the same way. The PCIe-EVR-300DC will be mounted directly to a PCI slot of an Industrial PC.

EPICS

The software integration is implemented through EPICS, specifically with the *mrfioc2* [33] EPICS module. *mrfioc2* was developed by the EPICS community specifically to integrate the different MRF products. ESS will be the first large facility to make an extensive use of the mTCA-EVR-300(U) and mTCA-EVM-300 boards, so during the development of this thesis several contributions were made to the *mrfioc2* source code, which will be present in the next release. The most important of these contributions are presented in Chapter 3.

mrfioc2 includes several parts needed for the timing integration:

- Kernel driver: it is the interface between the EPICS layer and the hardware layer. It recognises the EVRs and EVMs through their PCI ID and exposes them to EPICS and the software layer.

- Register map: it is a list with the registers of the EVRs or EVMs and their internal structure. It is used by the rest of *mrfioc2* to configure the EVRs and EVMs as defined by the users and operators.

- Device support: it is used to implement the timing functionality into *mrfioc2* without changing the default records, and allowing them to be used as required for timing. It is used basically to hide from the records the specific characteristics of the timing hardware.

- Databases: they are collections of instances of records with their specific configuration to integrate the timing functionality. The way they are built depends on the specific hardware targeted by each database.

Explained in a nutshell, the users and operators of the timing system will interface to the timing system by writing and reading EPICS process variables (PVs). The PVs are each of the configuration units of the records included in the databases. The device support reacts to the updated records by writing to and reading from the EVRs and EVMs registers through the PCI bus and kernel driver, by knowing the register map. The EVRs and EVMs will work as the configuration in their registers tell them. To use the *mrfioc2* module with the timing, the module must be loaded to an IOC with an (or several) EVR or EVM, alongside the related database. The EVR or EVM will be accessible from that IOC.

2.4.6 Timing system requirements

The most demanding requirements for the ESS timing system come from the data acquisition in some of the accelerating devices. Although not part of the timing system, the hardest requirements are put on the distribution of the RF signal along ESS, since all the accelerating devices need to be very well phase locked to each other. The timing system sends triggers to the RF systems, which react to those triggers on the following rising slope of the RF signal. This reduces the jitter requirements on the timing system, but still it should be good enough to guarantee triggering on a specific rising slope of the RF signal. Namely the jitter requirement for the ESS timing system is 1 ns, although the MRF timing system achieves much better jitter, in the range of a few tens of picoseconds RMS [34, 35].

There are a number of categories for the timing system requirements:

- Functional requirements
- Constraint requirements
- Interface requirements
- Graphical interface requirements

Some of this requirements are already fulfilled by the μTCA form factor, while others are by the MRF architecture, design, protocol and hardware. Following is a selection of the most important requirements for this thesis [36]:

- The timing system shall distribute timing to synchronize client devices in the accelerator, target, neutron instruments and conventional facilities.
- The timing system shall provide trigger signals to client devices to synchronise their operation. This includes pre-programmed sequences of timed actions and triggers.
- The timing system shall provide functionality to timestamp data that is collected from the client devices in such a way that the timestamps are synchronised all over the facility.
- The timing system shall be able to provide clock signals to client devices in μTCA form factor at the event frequency and integer subharmonics thereof.
- The timing system shall be able to transmit pulse type data to client devices, to enable client device configuration. The data delivery shall be time-deterministic and not affected by other on-going operations or network traffic.
- Actions on received timing events between two EVRs shall be synchronous, with phase jitter smaller than 1 ns.
- Resolution of timestamps shall be one timing clock period ($1/F_{event}$), i.e. 11.35686 ns.
- The jitter of clock and triggers distributed by the timing system shall not exceed 1 ns.
- EVRs shall be able to generate actions to received trigger events such as:
  - Physical pulses with configurable delay, width and polarity.
  - Notification to software components about the event reception.
- EVRs shall be able to inform any other software components running on the same I/O controller (computer) about the reception and the content of the received pulse data through a software interface.
- The timing system shall be scalable to support hundreds of receivers over the whole ESS facility, with (fibre) distances between components ranging from tens of centimetres to over kilometres.

- The timing system shall provide an EPICS interface as part of Control System integration. It shall be possible to configure and monitor the timing system behaviour over this interface.

- The timing system shall be phase locked to RF clock generated by the ESS master oscillator. The timing clock will be 1/4 of the bunch frequency, 88.0525MHz. The bunch clock signal shall be provided by the RF master oscillator.

- It shall be possible to remotely update firmware of all timing system hardware components. It shall be possible to verify the configured version and actual version of the system.

- EVRs shall be able to output timing system clocks and actions to timing events without an EVG (standalone operation mode only for testing purposes).

2.4.7 Timing system consuming systems

The ESS timing system is in charge of synchronizing the different parts of the ESS facility that require synchronization. Specifically the timing system only cares about the systems that need synchronization to create the proton beam with the desired characteristics, always within the design parameters, and the systems that need to be synchronized to the beam, such as the target wheel or the neutron instruments. On top of that the timing system will define the time reference for other systems and the conventional facilities, so from all the timing system functionalities only the common "real world" time is of interest to all the ESS facility.

On top of that there are other deliverables related to timing and synchronization but that are not in the scope of the timing system. The first one is the RF signal distribution that we have already talked about. Another one is the synchronization to the GPS signal, which in fact is an input to the timing system. The infrastructure team at ESS will take care of synchronizing to the GPS satellites with an antenna and providing NTP services based on that (to which the EVG synchronizes when it starts its operation). For the timing system and the ESS master oscillator also 1 PPS signals are needed. There are a number of systems that do not get synchronization by the timing system but from the NTP server, such as the PLCs, motion control and other EtherCAT based systems. There will be no PTP services

at ESS.

Regarding the systems that do get synchronized by the timing system, here is a list with the most important of them:

- Neutron detectors: they are the base of the experiments that will take place at ESS, since they provide the raw data that is extracted from ESS and that will be used by the researchers. They need timing synchronization to trigger the instrumentation and to be able to correlate the acquired data to the proton and neutron beams, since the delay of the neutrons, their speed, etc, may be important to the experiments.

- Neutron choppers: they select the neutrons that are needed for each instrument depending on their speed or energy. The neutrons that do not fulfil the required energy specifications are chopped away from the neutron beams. This is achieved by rotating discs with slots through which the neutron beams pass. By using several discs rotating at certain speeds at well defined spatial intervals it is possible chop away other neutrons, leaving only the desired ones. Timing synchronization is needed to synchronize all the rotating discs among themselves and to the proton beam.

- Target: it creates the neutron beams via the spallation process. The proton beam can only hit the target at very specific spots, and due to the very high inertia of the target wheel the proton beam needs to be triggered and accelerated to hit those spots. It needs synchronization to the timing system to rotate at the correct speed to provide neutron beams according to the parameters.

- Low Level RF (LLRF) and RF: the LLRF signal is used as a reference by the accelerating devices to create the RF signal that accelerates the proton beam. The RF signal is created by modulating the high voltage pulses provided by the modulators in the klystrons (amplifiers), using the LLRF as input signals. Timing synchronization is needed to trigger the modulators and feed the LLRF signal to the klystrons at the precise times.

- Ion source: it produces a low energy proton beam by ionizing a hydrogen gas and extracting the proton beam. It needs timing synchronization to extract the beam at the specific time so that, after being accelerated by the accelerator, it hits the target wheel exactly at the centre of each section.

- Beam choppers: due to the characteristics of the beam extracted from the ion source, the proton beam needs to be created longer than the final beam and specific parts of the beam need to the chopped away. This is done by the beam choppers. The beam choppers can only chop away a limited amount of the beam before being damaged. Timing synchronization is needed to correctly shape the beam.

- RF Local Protection System (RF-LPS) and MPS: they make sure that no damage is done to the machine, for example by stopping the beam when it may damage some equipment inserted into the beam path. The protection systems make sure that all the beam parameters are compatible with each other and that during operation they are not exceeded. For example, when running the beam in a specific mode with a limited beam length range, it measures that the actual length of the beam does not exceed what is accepted by the beam mode. The protection systems need timing synchronization to be aware of the beam parameters.

- Proton Beam Instrumentation (PBI) and beam diagnostics: they measure the beam along the accelerator to make sure that it has the expected characteristics. They work closely with the protection systems. They need timing synchronization for triggering and knowing the expected beam.

2.4.8 The ESS timing structure

ESS was designed with some well defined specifications such as the neutron flux, proton beam current or the proton beam power. Some of them relate to timing and will be explained in this section. All of these timing specifications are fulfilled by the timing system, which must take them into consideration for the design of the event timeline. All of these specifications will be derived from the RF signal of 352.21 MHz via the event clock of 88.0525 MHz. The event clock period, which is used as the granularity of the timing system, is approximately 11.357 ns long.

The basic frequency of operation of ESS is the beam pulse repetition rate of approximately 14 Hz that will create each of the beam cycles. This is the frequency at which the proton beam pulses are generated, and also has a strong dependency to the shape of neutron flux created at the target. Because of that the neutron instruments and experiments are also very conditioned by this frequency. Almost all of the accelerator devices are

triggered at this frequency, such as the Ion Source, the modulators or the choppers, although with different delays, to compensate for the beam time of flight and the own device's characteristics. This repetition rate will be generated at the EVG by counting exactly 6289464 ticks of the event clock with a multiplexed counter, so that the actual beam repetition rate will be 14.000000636 Hz, with a period of approximately 71.43 ms. The multiplexed counter will trigger the event sequencers that include the events that will trigger the accelerator devices.

Each beam cycle will include only one, well defined beam pulse, which is the window of time where proton acceleration takes place. The name is also used for the actual group of protons that are generated and accelerated during that window of time. The length of the beam pulse is approximately 2.86 ms. Because of the characteristics of the beam created by the Ion Source, the beam pulse is actually created longer than 2.86 ms, but chopped to this size early in the accelerator by the beam choppers. Because of the strict timing defined by the target wheel and its large inertia, the beam pulse window needs to always be at exactly the same point of the beam cycle. Nevertheless it is possible to have a shorter proton pulse inside the beam pulse window, and move that short proton pulse within the window, in practice moving the shorter proton pulse inside the beam cycle, but with strong limitations. This will be used intensively during the commissioning phase of the accelerator and target, and during ramp-up after the facility's shutdown periods. The physical beam pulse will be defined using the MEBT choppers which are triggered by two timing events exactly 251830 event clock cycles from each other, so the actual beam pulse length will be 2.859998 ms. Figure 2.8 shows the structure of the beam pulse in different points of the accelerator. These events will be defined in the event sequencer.

Another important timing structure is the bunch, although it is not created by the timing system but by the RF signal inside the accelerating devices. This structures happen naturally due to the oscillating nature of the RF signal used to accelerate the protons. The protons "surf" the RF waves in such a way that the protons in a specific position of the wave get certain acceleration, the protons ahead of the previous ones get less acceleration while those behind get more. The consequence of this is that the protons cluster together forming the bunches. Bunches are separated in time by one period of the RF signal, that is 2.84 ns. The spatial separation related to those 2.84 ns depends on the energy of the beam in each point of the accelerator.

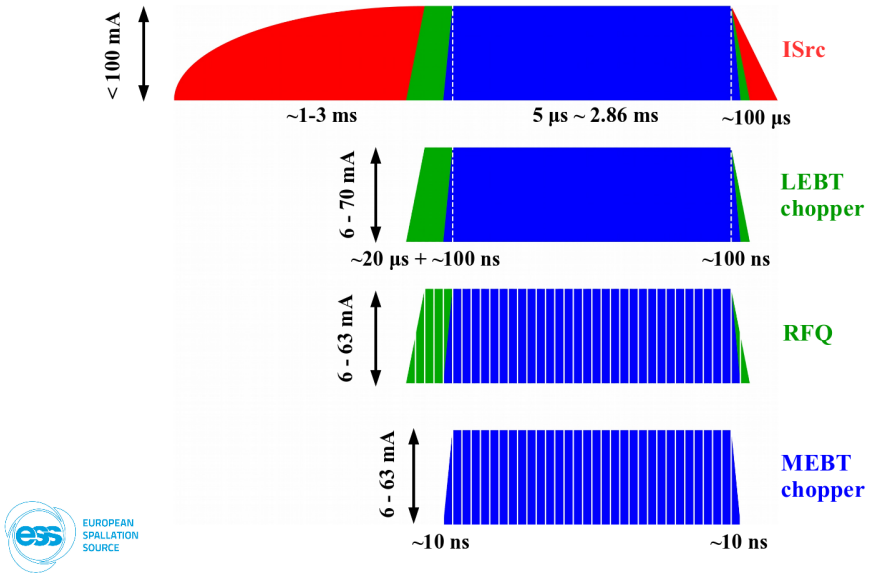

Figure 2.8: The beam pulse structure of ESS after several devices of the accelerator. Source: ESS.

Chapter 3

Hardware/Software codesign of the ESS timing system integration

Although the MRF timing system has been extensively used in other facilities, with very good performance, ESS needed a series of additional features to be able to overcome all the challenges to fulfil the requirements in order for ESS to become the world's most powerful neutron source. Some of these features were implemented by MRF in their products by ESS' request, such as the availability of EVMs and EVRs in μTCA form factor, of which ESS will be first extensive consumer, or at least one of the first large users from day zero, such as the delay compensation. Some other future developments, not yet available, were also requested from ESS and will be implemented by MRF in the near future, such as delayed gated signals triggered from events and μTCA EVR backplane inputs.

Some other small contributions were carried out by us during the development of this thesis [37], which among others include:

- Integration of the delay compensation mechanism into *mrfioc2*.
- Integration of the mTCA-EVR-300(U) board into *mrfioc2*, including support for the new backplane clock lines.
- Integration of the mTCA-EVM-300 board into *mrfioc2*.
- Integration of the *mrfioc2* module, including the kernel module, into E3, the new ESS EPICS Environment.

- Preparation of start-up scripts for all timing system boards and all their functionalities.
- Debugging of the timing system. One important debugging process took place with the old VME-EVR-230RF, used before the mTCA-EVR-300(U) was available. The integration of this board had important interrupt issues that needed to be fixed. The problem was in the PCI-to-VME bridge module used in the VME[1] system used to deploy the EVR.
- Implementation of a hardcoded-free naming structure in *mrfioc2*. The *mrfioc2* EPICS module included the FRIB naming convention, hardcoded in some places, by default. This made it difficult to quickly deploy *mrfioc2* in a new facility with a different naming convention, since a lot of changes were needed. We have changed the *mrfioc2* source code so that the naming convention is only revealed in the substitution files that are inflated to create the EPICS databases. This makes it really easy to deploy *mrfioc2* at a new facility, since all of the changes are encapsulated in the substitutions files. Also the substitutions files with the ESS naming convention were developed.
- Deployment and maintenance of a central EVG and timing distribution with basic 14 Hz events in the ESS controls lab.
- Support for ESS and in-kind partners' engineers setting up and maintaining their timing systems. Also helping them interfacing their equipment to the timing system.
- Preparation of technical manuals, trainings, workshops and other documentation for ESS engineers.
- Study, implementation and deployment of the ESS timing distribution, including the physical deployment of timing crates, fan-outs, fibres and the EVG with its necessary inputs from the RF master oscillator and GPS receiver.
- Configuration of the timing distribution crates. This includes the physical configuration of the crates and the AMCs and the functionality configuration of the MCH[2] and the Concurrent AMC processor, the EPICS layer and IOC start-up scripts.

[1] Versa Module Eurocard bus.

[2] MicroTCA Carrier Hub. It manages the crate, from the power distribution to the AMCs to routing the connections and monitoring the whole crate.

In this chapter we present some of the most important contributions carried out for the ESS timing system during the development of this thesis. In the second part of this chapter we present the data model specification that was developed for the ESS timing system during this thesis. The data model specification is the list of events and the list of items in the data buffer that are needed to fulfil the ESS needs. This is basically done by broadcasting all the necessary deterministic information and the events that allow all different systems to be configured and triggered correctly to run ESS in a successful manner. Creating the list of beam parameters and events, the relations among them and how they are used by each system is an indispensable milestone of the design of every accelerator-based research facility.

3.1 Standalone mode

One of the early requirements for the ESS timing system, basically due to the in-kind nature of the ESS project, where in-kind partners all over Europe develop and produce different ESS sub-systems, was that the EVRs should be able to work in an autonomous way. This is needed to allow designing and testing the different sub-systems with a timing interface and signals all over Europe without deploying a full and very costly timing system (consisting at least of one EVR and one EVG), even if it is with reduced expectations on the requirements. In normal conditions EVRs can not work at all without the data stream coming from the EVG, so a different approach was provided to allow them having a timing interface and signals. In the newly implemented standalone mode in the mTCA-EVR-300(U) and PCIe-EVR-300DC EVRs firmware can use their TX port to send a data stream to the RX port via a loopback optical cable. In this case the EVR is configured to use the fractional synthesizer frequency, which usually is used as a reference for the PLL, as the event clock. A sequencer, identical the ones found in the EVGs, has being included in the EVR firmware to allow the generation of events. The sequencer is usually triggered by a prescaler, but it can also be triggered manually by software, by the pulse generators, or the distributed bus clocks. The standalone mode is usually used by triggering the event sequencer at 14 Hz, mimicking the beam cycle, and sending different 14 Hz signals to the physical outputs. Harmonics of the 14 Hz beam cycle frequency are usually also implemented.

Another use of the EVR sequencer is triggering a local sequence of events

from one of the delay generators, which at the same time is being triggered by an event from the EVG data stream. This allows the EVR to have a series of local events with a tight synchronization to the events from the EVG. The neutron choppers use this functionality extensively: in their case, several integer multiples of the 14 Hz beam repetition is needed. Instead of using several of the only eight available distributed clocks or a number of repeating events, and since they need to be perfectly locked to the 14 Hz master repetition, these frequencies are generated locally with a sequence that is triggered by the 14 Hz master event. Having the exact same sequence in different EVRs, and thanks to the deterministic and very precise computation on the hardware level, it is even possible to guarantee that different EVRs are locally generating the same events at the exact same time. This allows the synchronization of neutron choppers, sometimes tens or even hundreds of meters away from each other, in a safe way, with multiple synchronized clocks, and using only a minimum of resources from the bottle-necked data stream from the EVG.

During the work carried out for the development of this thesis the need for the EVRs running independently from an EVG was identified, refined, defined as a set of technical requirements and communicated to MRF. After the firmware implementation was developed by MRF and sent to to ESS we also implemented in *mrfioc2* the new standalone mode, tested and debugged it, and prepared documentation and examples for the in-kind partners and other engineers at ESS. We also supported all of them setting up the standalone EVRs in their projects.

3.2 The miniIOC

The ESS neutron beamlines have a specific instrumentation for monitoring and shaping the neutron beams according to the needs of each specific experiment, such as neutron detectors and neutron choppers. The neutron beamlines are special environments that have tight requirements on the timing system, mainly on the hardware form factor. For this reason the EVR hardware shown in Section 2.4.4 is not the most suitable for this application. ESS has decided to sign an in-kind agreement with the University of Tallinn in Estonia, which will provide ESS, in the form of an in-kind project, an embedded controller that will fit the beamlines applications. This embedded controller or miniIOC should be light and compact, in DIN-

rail mountable form factor, and be based on the Zynq System-on-Chip [38] (Soc), which includes on a same silicon die a FPGA[3] and an ARM [39] CPU[4]. The FPGA is needed for a better control on the time domain of the events and signals, and a very high processing speed possible by the hardware computations. FPGAs can also synchronize their internal working frequency with an externally-supplied high precision frequency, which for the miniIOC operation will be the event frequency of ESS. The CPU will be used to run embedded Linux, including the EPICS layer used for interfacing the control system. The miniIOC should offer high reliability and be remote manageable, with low power consumption, passively cooled only and include no moving parts such as cooling fans.

The main interface of the miniIOC to the neutron beamlines will be a LPC FMC (low pin count FPGA mezzanine card) to allow for great flexibility in the actual interfaces, driven by the fact that the hardware of the beamlines is not standardized, and thus the connections are not either. The FMC connector allows the miniIOC to be compatible with whatever connectors the beamlines need in a very easy way, just by swapping the FMC board, which is basically an input/output mezzanine module. The FMC is connected directly to the FPGA, achieving low delays and very good quality signals. This provides the miniIOC with re-configurable input and output capabilities.

One of the uses of the miniIOC is to include an embedded EVR. As explained in Chapter 2, one of the characteristics of timing systems, of which the EVRs are part of, is that they are able to provide very tight synchronization with low jitter and high temporal accuracy, while maintaining a great degree of determinism. Processors and software, even with real-time kernels, are not good candidates for the timing and synchronization use-case, for that reason, the timing modules are implemented in FPGAs, and also in the miniIOC the EVR functionality will be implemented in the PL (programmable logic) section of the Zynq board. The PL is basically an integrated series-7 Xilinx FPGA, which is the same family as the FPGA used for the μTCA and PCIe EVRs (Kintex 7).

Most EVRs only use a reduced amount of the available resources: even though, for example, the mTCA-EVR-300(U) has eight 32-bit prescalers and

[3]Field-Programmable Gate Array.
[4]Central Processing Unit.

sixteen pulse generators (with different prescaler, delay and width ranges), most EVRs only use a subset of them, basically because most triggers are shared through the eight backplane trigger lines. The reason for having so many available resources is that the EVRs only have the responsibilities related to timing, as explained in Section 2.4: event distribution, generation and distribution of synchronous clock signals, definition of a common time base and timestamping functionality, and transmission of fast and synchronous beam-related data. No more functionality is needed from the EVRs, and a big amount of resources is included for flexibility (making use of the RTM outputs, etc).

The miniIOC, on the other hand, may include other functionality in the FPGA, such as data acquisition, compression, processing or transmission, all of them activities which usually are performed by the FPGA logic [30]. In that case it will be desirable to implement in the FPGA logic only the timing resources required by that specific miniIOC implementation, leaving free for other activities as many resources as possible.

The other part of the Zynq board is the processing system (PS), the CPU of the system, where the configuration and integration into the rest of the control system of the embedded EVR is implemented. These are activities that usually are performed in the software layer, and that for this reason are better suited to a CPU. The PS is capable of running embedded Linux and input/output interfaces.

This makes the Zynq SoC a very good candidate for the core unit of the miniIOC, since it combines the high reliability, high determinism and high configurability characteristic of the FPGA with the high flexibility of the CPU.

The in-kind project includes the development of the hardware board, which will be delivered to ESS as a number of finished physical boards, including the hardware schematics, but not the software or firmware (FW) image or bitfile with the embedded EVR functionality, which is needed for the intended use of the miniIOC board. The software kernel will be embedded Linux, with a root file system that includes the EPICS environment and other appliances to integrate into the ESS timing system.

On top of the FMC connector, the miniIOC should include a SFP cage for interfacing the timing system and receive the timing bitstream from the

EVG and an Ethernet interface to integrate the miniIOC into the ESS control system, remotely manage it and boot from network, since it will include no permanent local storage for the Uboot, Linux image, FPGA image, etc; all of these should be loaded from an external server.

The firmware should be developed in-house at ESS, complying with the requirements put on the miniIOC, mainly the high configurability that is expected from the form factor. The existence of the FMC also means that the input/output should be highly configurable. Even more than that, the FMC can be used to acquire some data in real time, which may be processed by the FPGA before being sent somewhere through EPICS, for example for archiving. This means that the EVR footprint in the FPGA should be as small as possible while at the same time fulfilling all the requirements. The best way of providing this is then using customized embedded EVRs for each application, that only include the functionality needed for that specific application, omitting all other non-needed timing modules. This imposes new desired characteristics on the bitfile source code: it should be easily configurable, at least in the number of instances of prescalers, pulse generators, etc, according to the miniIOC implementation.

The miniIOC should be able to not only be configured with a new bitfile easily, but also it is expected that the bitfiles will need to be changed and synthesised easily, quickly and as effortlessly as possible. Since most engineers (even firmware engineers) do not have much experience with timing systems, it would also be desirable that the implementation and synthesis workflow is as easy and high-level as possible.

All of this means that the miniIOC FPGA configuration, which is done through a bitfile, and by the extension the VHDL source code that is synthesized and compiled to obtain the bitfile, needs to be generated in an easier, faster, and less error-prone way than by a FPGA engineer writing, validating and testing each of the possible configurations that may be needed in each of the miniIOCs deployed in the ESS facility. We have developed a method to automatically synthesize statecharts, an extension of Finite State Machines, that represent an EVR, into synthesizable VHDL code from only a graphical description of an EVR represented as a statechart (a description of statecharts is presented in Section 4.1). Although this is the triggering use-case, the method can be used to generate code to any application, not just EVRs, as long as they can be represented as statecharts. This application is presented in Chapter 4. This automated methodology also keeps the

chance of errors as low as possible.

A proof of concept of the miniIOC was developed by the author of this PhD thesis as a Master thesis. That proof of concept included the study of the MRF timing system and its specification, the compilation of the requirements for the miniIOC and the search for a hardware architecture that covered all the requirements. Then the implementation of the software layer, from the operating system installation to EPICS and kernel driver compilation was performed. For the proof of concept the FPGA resources and specifications were checked, but no actual HDL[5] programming was implemented.

3.2.1 Embedded EVRs

Although we have presented our application as the chosen way to configure the miniIOC, it may have other uses. One of them is to configure embedded EVRs. Embedded EVRs are independent blocks of code, with a clear interface, that implement the functionality of a EVR. They will be synthesized and compiled for any other FPGA that does not have timing as its main application, but that in any case needs some timing information (assuming that the board containing the target FPGA also has a SFP module that can connect to the timing network). The traditional way of providing timing to these boards is by sending the signals through the front panel or backplane (such as in a μTCA system) from an EVR that receives and decodes the timing bitstream from the EVG, but assuming there are enough free resources on the FPGA, and that the hardware supports the timing link, it is possible to decode the timing information and react to it in any FPGA. In this case it is even more important to have an embedded EVR that has the smallest footprint possible, since most resources should be kept for the main application of the FPGA. Because of that, being able to customize to the maximum extent the embedded EVR, using only the minimum needed resources, is very important. This also means that each application will have a customized embedded EVR, so in this case being able to configure, change, implement and compile the embedded EVR code as easily, quickly and effortlessly as possible, while minimizing the chance of errors and bugs in a very large VHDL file, is even more important.

[5]Hardware Description Language.

The application presented in Section 4.1 can also be used to generate embedded EVRs, although this usecase has not been implemented yet.

3.3 The supercycle

For commissioning purposes, a high level application is needed that allows changing the configuration of the timing system rapidly but in a safe way, and thus the operation of the ESS machine. This application is currently under development and will allow the operators to pre-configure a series of sequences that will run automatically when triggered. For this reason this high level application is called supercycle.

The supercycle application is how the operators control the ESS machine when it is not running in a steady state. It is the main way of changing the configuration of the EVG, which at the same time controls the operation of all ESS parts and synchronises them.

The supercycle application is in charge of defining the events and items in the data buffer that are needed for the correct operation of ESS (the synchronous clocks and the broadcast of the common sense of time or timestamping need to be correctly configured but they do not need an active configuration from the operator during operation, but rather are configured once at start-up and then they work on their own without intervention). The supercycle application must, on the one hand, write the correct sequence to the sequencer. This includes making sure that all the needed events are included in the sequencer and that have a correct timestamp and that they are ordered in ascending order according to their timestamp. On the other hand it must know what information is needed in the data buffer, where to get this information from, and make sure that the information is correct. It also needs to trigger the broadcast of the data buffer at the correct moment. Since both the event list and the data buffer are going to be different pulse-by-pulse at the operation frequency of 14 Hz, which is obviously a frequency too high for a operator to manually change the configuration in real time, the supercycle application should be able to store in memory a big enough number of the configurations to satisfy the commissioning works when the different beam configurations are run in automatic way. The supercycle application is then triggered either manually for a single run or can be run in loop mode. The supercycle application needs to be able to work

in a synchronous way with the operation frequency of 14 Hz, making sure that the new configuration for the next cycle is loaded in the EVG after the previous sequence has finished but before the new sequence is triggered, and also the data buffer should be written and triggered at the precise time inside the beam cycle. Finally the supercycle application is also expected to cross-check the information that it gets from all sources and make sure that all this information, and the parameters that is writing to the event sequencer and the data buffer, is consistent.

The supercycle application will consist of four parts:

- An operator interface with graphical user interface: this is how the supercycle application gets its configuration parameters. It should not be a simple graphical interface for the EPICS layer, but rather it should have a basic set of parameters that easily but uniquely describe the beam, and that is a natural way of representing the beam for a human operator.

- A translation engine: translates the parameters introduced by the operator to a correct sequence and data buffer, according to the implementation of the timing system. This is basically a set of EPICS waveform records, two to represent the event sequence (one with the events and a second one with their timestamps, both ordered in ascending timestamp order) and one for the data buffer.

- A memory to store the configurations.

- A synchronous sequencer and data buffer writing mechanism: it should make sure that both the sequence configuration and data buffer configuration are written to the EVG at the correct time and that they are triggered synchronously to the beam.

The supercycle application is not only used for commissioning, but can be also used as a ramp-up application. Although the actual implementation of the supercycle application was not developed during this thesis, and similarly to the standalone mode, the need for it was identified and the list of requirements was compiled by us. We also designed the interface between the supercycle and the timing system and helped designing the translation engine and the integration of the synchronous sequencer and data buffer.

3.4 The ESS data model specification

One of the things needed to run the ESS facility successfully is triggering all the devices exactly when it is needed so that they working in a synchronised way, and the timing system is the responsible for the triggering. The ESS timing system uses events to generate the triggers that are necessary along the facility, and the events need to happen with very specific, timed delays among them. Developing the list of events that can successfully trigger all the systems of the machine is not an easy task that needs to be done from scratch at every facility, since the characteristics of each machine are unique. These characteristics include the specific systems that make up the facility (for our concern these are the consuming systems of timing), their configuration, their interfaces and also their physical location, as the triggers need to take into account the time of flight of the beam [40].

In parallel to the list of events, and for similar reasons, it is necessary to have a list of the beam parameters data that are needed by all the systems, so that this information can be included into the timing data stream. The study and development of both of these lists, as well as the rationale to put events or data item in one or the other, has been performed as part of this thesis.

This information has been compiled in an internal ESS document *Data Model Specification for the ESS Timing System* [41].

3.4.1 Differences between Event and Data

The timing system provides information to the ESS facility by using two different means: events and the data buffer. Different kinds of information are usually better suited for one of the two means. The characteristics of events and data in relation to the information that they provide are:

- Events are propagated from the EVG to the EVRs in an accurate and deterministic way and can directly trigger hardware actions. There is no restriction on the repetition rate, etc, other than only one event can be sent every tick of the event clock. There is a limited amount of events (256, some of them reserved) and the purpose of any of them has to be pre-defined so that they can be used in the timing system. A number of sources can trigger event generation: sequencers, multiplexed counters, hardware inputs and writing to a register are the

most common; all of the sources can be active at the same time, with a priority encoder for resolving simultaneous generation of events.

- Data is transmitted to the EVRs by a sequence of first preparing a data buffer in the EVG and then requesting a transmission. Data is transmitted byte by byte to the EVRs and once the full buffer has been received, the EVR notifies the system CPU by sending an interrupt. The transmission is typically done once per pulse. It is available as an EPICS waveform record, so all processes related to it happen in the software layer, and thus are only deterministic to the level provided by the software; in this context typically to tens of microseconds. The data transmission is up to two KiB in size and completely configurable, even pulse-by-pulse. It is possible to use the data buffer to send complex information.

With these characteristics, a policy for distributing the information sent by the two means has been defined:

- Information that requires strict determinism on a sub-nanosecond level or is delivered as digital signal or a sequence of digital signals is sent as events.

- Information that describes a beam pulse parameter, represents a physical magnitude or is better described as a float or integer value is included in the data buffer.

Data is sent for configuration and/or confirmation purposes and shall not be used as triggers.

3.4.2 Operation event list

In this section the events that are used in the ESS sequencer are listed and explained. This list was implemented as part of thesis:

- Start of cycle: it is used as the master event, running at 14 Hz, that sets the start of a new cycle. It is always present, even when there is no beam, to allow keeping a periodic reference. Is is also needed for systems that cannot loose synchronization at any time or that, due to several factors, are slow and cannot resynchronize quickly to an intermittent beam cycle frequency, such as the target wheel. It needs to be sent as soon as the sequencer is triggered at beginning of

the sequence. All of the ESS systems need to be aware of the master frequency defined by this event. Event number: 14 (0x0E).

- Ion source start: it triggers the magnetron of the ion source, so that the extraction of protons starts. It is always sent, even when running at slower frequencies than 14 Hz, to keep the stability of the beam, and only stop if a larger break of the beam pulses is expected. It should be sent at least 3 ms before the actual 2.86 ms beam pulse is expected, due to the characteristics of the extracted protons. It is used by the ion source to trigger the magnetron. Event number: 10 (0x0A).

- Beam pulse coming: this is a pre-trigger for several systems that need to prepare before the beam arrives. It is present in the sequence whenever LLRF needs to be triggered, this is, when the accelerator needs to be triggered because it is in running mode, even if the current beam cycle is empty, for the thermal stability of klystrons, moderators and other devices. If there is no beam pulse in this cycle (if it is empty) this is marked by an item in the data buffer for the systems that need to know if they should expect beam, such as the Beam Current Monitors (BCMs). It should be sent at least 300 μs before the actual 2.86 ms beam pulse is expected. It is used by the LLRF systems to start filling the RF cavities before the beam pulse arrives (ramp-up) and by the BCMs to start the monitoring process. Event number: 15 (0x0F).

- Beam pulse start: it marks the start of the beam pulse and triggers the choppers that will shape the beam pulse. LEBT and MEBT choppers both use this event with different local delays. It is triggered under the same conditions than the "Beam pulse coming" event. The time relation between this event and the "Beam pulse coming" event has to be respected (300 μs). This event is used by the RF systems to compensate for the loss in the energy filling of the cavities when the beam pulse arrives, by the LEBT and MEBT choppers to shape the beam, and by the BCMs to know how is the expected beam and compare it to the measured one. Event number: 12 (0x0C).

- Beam pulse end: it marks the end of the beam pulse and triggers the choppers that will shape the beam pulse. As with the "Beam pulse start" event, LEBT and MEBT choppers use different local delays. It is triggered under the same conditions than the "Beam pulse coming" event. This event is used to stop the RF systems, by the LEBT and MEBT choppers to shape the beam, and by the BCMs to know the

expected end of the beam and compare it to the measured one. Event number: 13 (0x0D).

- Ion source end: it deactivates the magnetron of the ion source to stop the extraction of protons. It is sent whenever the "Ion source start" event has also been sent. Event number: 11 (0x0B).

- Send data buffer: it is used to trigger the transmission of the data buffer so that it is synchronized to the beam cycle and the devices can be ready before the next beam pulse. It has to be always present, since the data buffer includes important information such as if beam is expected in the next cycle. It is used by the supercycle application to send the data buffer from the EVG to the EVRs. Event number: 16 (0x10).

- Post mortem event: it tells participating systems that they need to save and deliver data around the time (before, during and after) of a beam stop, to be able to examine and learn from the situation where a beam stop took place. It is not sent from the sequencer, but raised when requested by MPS or PBI, that will also be the consumers of the event in the rest of the facility. Event number: 40 (0x28).

- Data on demand: it triggers data acquisition in multiple subsystems simultaneously, for collecting coherent, synchronized data sets. It works like a trigger signal for a distributed oscilloscope. This event can be activated with a user request or by a client system by sending a signal to an EVR in the subsystem (from that EVR the signal will be propagated upstream to the main EVG and from there back to all downstream EVRs). Event number: 41 (0x29).

- BCM calibration: it triggers the calibration of the BCMs by marking a time slot where only the background will be measured. It is sent from the EVG at periodic intervals in the range of several minutes. Event number: 42 (0x2A).

Event timeline

Figure 3.1 shows how the events triggered with the sequencer will look like when ESS is running in steady operation. This sequence of events and their delays from the start of the sequence was written by us in the form accepted by the timing system and *mrfioc2* and is being used by some test systems

already.

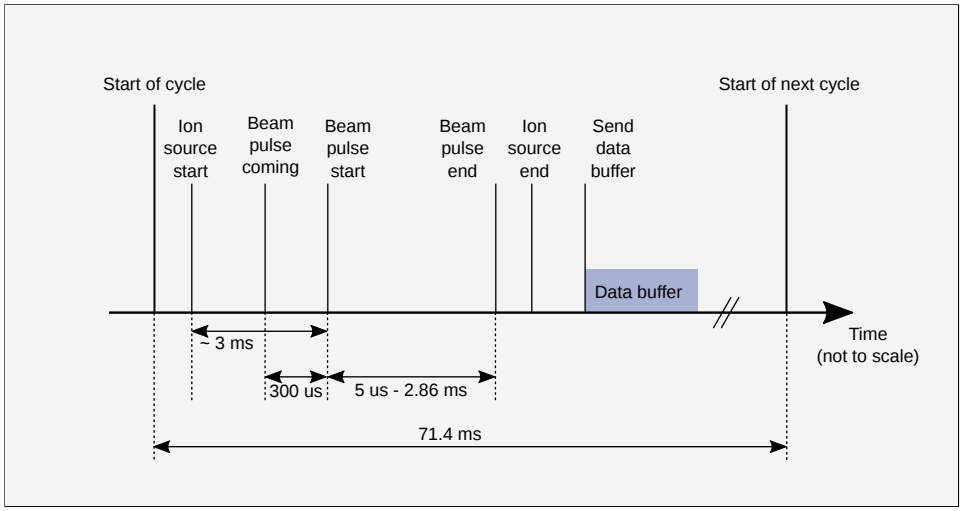

Figure 3.1: Example of the timeline of the events in the sequencer during normal operation.

Reserved events

There are a number of events reserved for internal use of the timing system, which are listed here [42]:

- Event 0 (0x0): null/idle event.
- Event 112 (0x70): timestamping related event, shift in '0' to less-significant-bit of Seconds Shift Register.
- Event 113 (0x71): timestamping related event, shift in '1' to less-significant-bit of Seconds Shift Register.
- Event 121 (0x79): stop event log event, stop writing events to event log.
- Event 122 (0x7A): heartbeat event, reset Heartbeat Monitor.
- Event 123 (0x7B): synchronise prescalers event, reset all EVR dividers.
- Event 124 (0x7C): increment timestamp counter event.

- Event 125 (0x7D): reset timestamp counter event.
- Event 126 (0x7E): beacon event, delay compensation signal.
- Event 127 (0x7F): end of sequence event.

Event number allocation

The event codes are distributed depending on what they are used for. There are events used for triggering devices according to the event sequencer, special asynchronous events, local EVR input events (mostly used to timestamp signals) and local EVR sequencer events (used to create local frequencies at the EVRs synchronized among them or to the master 14 Hz frequency). This is how event numbers are distributed depending on their use:

- 1-39 (0x01-0x27): repetitive events for beam operation, in the master sequencer.
- 40-49 (0x28-0x31): special non-repetitive events (post-mortem event, data-on-demand, etc).
- 50-59 (0x32-0x3B): other master events (multiplexed counters, software events, etc).
- 60-111 (0x3C-0x6F): not allocated.
- 112-113 (0x70-0x71): pre-defined special events reserved for system use.
- 114-120 (0x72-0x78): not allocated.
- 121-127 (0x79-0x7F): pre-defined special events reserved for system use.
- 128-149 (0x80-0x95): local EVR input events.
- 150-199 (0x96-0xC7): local EVR sequencer events.
- 200-255 (0xC8-0xFF): not allocated.

3.4.3 Data definition

In this section the items that are included in the ESS data buffer are listed and explained. This list was implemented as part of thesis:

- Protocol (number and version): the protocol refers to the data items and their organization inside the data transmission buffer; this allows for different information contained in the data transmission and to be able to know its exact structure for each transmission. It has a length of 2 bytes; the first byte represents the protocol number and the second byte represents the version of that protocol. At the present time only one protocol is planned, with as many versions as needed. The protocol number and version are issued by the timing system and consumed by all systems.

- Beam pulse ID: it is used to identify with a unique, integer number each beam pulse against the recorded timestamps. It is expressed as an 8-byte unsigned integer to allow unique IDs during the whole lifetime of ESS incrementing at 14 Hz. The ID increments monotonically by one for each new beam cycle. Empty beam cycles will also have an ID. It is issued by the supercycle application and used by all systems.

- Beam present: it marks if beam is expected in the next cycle, so that different systems can react to the beam or raise flags and signals if the beam happens in a different way to what is expected. It is expressed as a boolean (0 no beam, 1 beam present) although transmitted as one 8-bit byte to maintain byte alignment in the buffer. It is issued by the supercycle application and used by the BCMs to know if the cycle contains protons in order to react appropriately (monitor beam parameters) and by LLRF to react to the beam.

- Beam destination: it is the planned destination (last point in the path) of the proton beam, where it hits. In normal operation it will be the target wheel, although during commissioning phase and ramp-up of the accelerator other destinations are possible. The destination is encoded in a 8-bit enumerated integer, with the possible destinations and their code defined in Appendix B. All other values are to be treated as an error. The beam destination is issued by the accelerator group and used by all beam PBI systems.

- Beam mode: it is the planned beam mode as defined in Appendix A. Different beam modes are used to specify some constraints to the accepted beam pulse. For commissioning and ramping up the accelerator

some special, very sensitive devices are inserted into the beam path to measure it with higher. These are called insertable devices and can only accept beam pulses with certain constraints, which are defined by the beam modes. Beam modes are also used to ramp up the accelerator in a slow, controlled way so that it does not suffer damages. The beam mode in encoded in a 8-bit enumerated integer. All other values than those shown in Appendix A are to be treated as an error. The beam mode is issued by MPS and used by all the PBI and beam diagnostics devices, both for checking that the expected beam is safe for the state of the accelerator and for raising protection signals if it is not.

- Intended pulse length: it is the planned length of the beam pulse expressed in milliseconds with a 4-byte float. Possible values are between 5 μs and 2.86 ms. It is used by PBI and beam diagnostics devices to check that the beam pulse has the desired characteristics. It is issued by the supercycle application.

- Intended proton energy: it is an estimation of the planned proton energy at the beam destination expressed in Mega-electron-Volts (MeV). It is a 4-byte float and is used by the target to estimate the thermal effects on the target wheel. It is issued by the control room operators.

- Intended beam current: it is an estimation of the planned beam current at beam destination expressed in mA with a 4-byte float. Similarly to the intended proton energy it is used by the target to estimate the thermal effects on the target wheel. It is issued by the control room operators.

- Raster pattern: it is an index to select one of pre-defined patterns. The raster patterns are the method used to distribute the proton beam over the whole surface of each of the sectors of the target wheel. To achieve this at the end of the accelerator there are raster magnets that bend the proton beam in specific patterns, the raster patterns. The different patterns are enumerated using a one byte integer. It is issued by the operators and used by the raster magnets.

- Target segment: it is the planned target segment for the next beam pulse. This allows the target to request to stop the beam if the sector is too warm. The planned target segment is represented by a one byte integer number where only numbers different than 0 are allowed when the beam destination is the target. If the beam destination is

the target, segment 0 means that the segment synchronisation has not been achieved and the beam should be prevented. Otherwise each sector has an associated number in the range from 1 to 36, and other values are invalid.

Data buffer item list

Table 3.1 shows a list of the organisation of the beam parameter items inside the data buffer. It includes the order of the items in the data buffer, how long they are and what is their offset from the beginning of the data buffer. Also an EPICS database that reads the data buffer in the EVRs and partitions it into meaningful pieces ready to be used by other devices integrated into the EPICS control system has been written as a part of this thesis.

Table 3.1: ESS data buffer item list.

Byte offset	Byte length	Item
0	2	Protocol Version
4	8	Beam Pulse ID
12	1	Beam Present
13	1	Beam Destination
14	1	Beam Mode
16	4	Intended Pulse Length
20	4	Intended Proton Energy
24	4	Intended Beam Current
28	1	Raster Pattern
29	1	Target Segment

Chapter 4

Automated synthesis of Statecharts

Some of the ESS components based on FPGAs, such as the miniIOC presented in Section 3.2, require an easy and error-free way of implementing their logical behaviour, which is done by loading a bitfile which has all the information necessary to configure the FPGA resources in the correct way to perform the desired tasks. The file is generated by compiling a source code file in a hardware description language such as VHDL. The usual procedure for generating the bitfile include a firmware engineer writing the VHDL source code, writing and running a battery of tests and simulations, and synthesizing, implementing and compiling the bitfile. All of this may be a very slow and error-prone process, and there is a lack of tools that help in this regard. For this reason, a tool that can automate this process while keeping it as simple and fast as possible is very desirable and has been developed in this work [43]. This tool would be even more useful when the system needs to be re-configured often with a different number of resources or instances of sub-systems or when the input/output can change, such as the miniIOC with its FMC connector.

The tool should not impose any extra requirements or limits on the application that will be implemented. To represent the target system we have chosen the statecharts, an expansion of Finite State Machines (FSMs) introduced by Harel [44] in 1987 which solves some of the problems of FSMs, mainly the exponential growth of states and transitions when the machine adds more and more parameters or conditions. Despite they are gaining traction, a reduced number of tools support statecharts. In the context of

systems control, an automated way to produce HDL code from a statechart is required in order to deploy a new timing system configuration in a short time without incurring in implementation errors.

Our tool takes this graphical, behavioural implementation of the system as a statechart and without any more interaction from the user it provides a synthesizable VHDL code file, sparing the user from implementing the HDL code by itself. The generated firmware configuration is ready to be deployed quickly in varying environments while keeping the chance of errors low.

Some parts of the EVR functionality do not fit well when expressed as statecharts, such as the timestamping mechanism, since due to their characteristics they would implement all their functionality in a single state. These parts would need to be coded by hand by a firmware engineer. Then the code produced by our tool and the hand-written one should be merged together. This will not be a problem for our target application since the parts that need to be written by hand will not need to change in new versions, so they can be developed once and used for the foreseeable future.

In this chapter we will present the statecharts in detail, look at the previous work done in the area of statechart hardware implementation, how we can translate the statechart graphical implementation in a conceptual way, how each of the statechart features can be implemented and how our tool works. Finally we study the possibility of updating the statechart implementation without requiring logical synthesis. This possibility is addressed later in Section 4.4 based on microprogramming [45, 46], where a method for implementing generic microprogrammed architectures is proposed [47]. Microprogramming would allow us to conduct upgrades of statecharts on the field just by loading a new control update that modifies the behaviour of the old firmware, without actually changing or synthesizing it again. The advantages and disadvantages of this approach are also evaluated.

4.1 Statecharts

Statecharts are a visual formalism for describing states and transitions, and they became part of the Unified Modeling Language (UML) [48]. As stated in Harel's paper, statechart diagrams were introduced with the intention of expanding the capabilities of FSMs. FSMs are state-based models where

only one state is active at any given time, which can be changed by external inputs or internal conditions. The change between states is called transition.

In Figure 4.1, a statechart is shown. At the `top` level, there is one *entry node* pointing to the state that gets active when the statechart starts, and two *OR*-super-states. Super-states are special kind of states that group other states together and the relations between them, which may even form complete FSMs inside the super-states. `active` and `wait` are super-states, specifically *OR*-super-states, because either `active` OR `wait` may be running at a given time, and they both have an inner structure that includes other states and transitions. It is possible to switch between `active` and `wait` using the `sleep` and `wake` *events*, which trigger the transitions. The `active` super-state is made up of two super-states or components, `send` and `receive`. This illustrates the concept of hierarchy, as one super-state may contain several other ones. In this case, both super-states are also running in parallel, allowing to describe concurrent processing. This is called an *AND*-super-state (denoted by the divider line). *AND*-super-states are called *regions* in the UML description, since they are slightly different from regular states: they do not have direct transitions, instead they are enabled or disabled only when the state including them is as well activated or not; also they are not affected by conditionals or actions, both explained later. Both the `send` and `receive` states include more states, and transitions between them. Some of these transitions are just triggered by normal events, such as `send` or `resume`, while others include special modifiers, such as the `send_once` condition or the `transmission = false` action. Transitions may also have delays or timeouts, such as the `always` modifier that sets a 0 delay (another example is the `every 10 s` in the nested `background` state inside the `wait` super-state). Some states may also trigger events, such as the `ACK` state, that triggers the `acknowledge` event when entering the state. In this case the `acknowledge` event does not cause any action in the statechart and is expected to be used as an output and consumed externally, but it could also trigger something to happen in a different part of the statechart. A black dot and an arrow point at the initial node for each super-state. In this example there are two objects, one state and one event, both called `send`, but they should not be mixed since they have different purposes and appear in different situations. The representation shown in this example is different from the one presented in [44], but the concepts are the same.

Contrarily, when the `wait` state is active, either `idle` or `background` are running, but not both at the same time. A black dot and an arrow point at

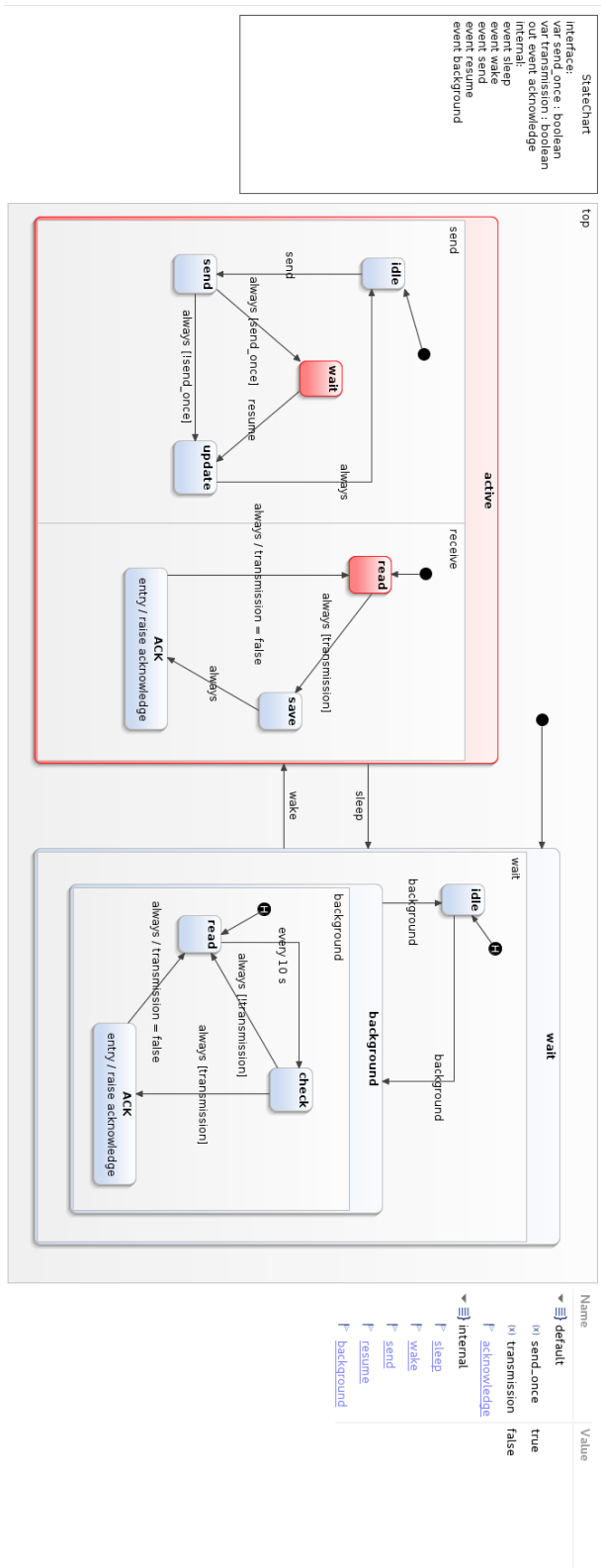

Figure 4.1: Example of a statechart in Yakindu SCT.

the initial node for each super-state. Inside this `wait` super-state, two other super-states are denoted to have *history* (an H within the dot). Therefore when processing returns to `wait` after it has been left, it remembers whether it was running in `idle` or `background` and, in the latter case, in which of the three nodes. In the example shown, at that precise time the `active` super-state is enabled while the `wait` super-state is disabled, as well as all their nested states. Inside the `active` state both `send` and `receive` states are active, and inside each of them one level down in the hierarchy, also one state is active: the `wait` state in `send` and the `read` state in `receive`.

The aspect that limits the usability of FSMs is that they can greatly grow in complexity when adding more states in bigger state machines. Statecharts were introduced as a way to specify complex systems in which there may be several sub-systems working in parallel (which are implemented in statecharts as states active at the same time in different super-states) or a large number of events and transitions to evaluate, such as the timing system in a big research facility. Statecharts introduce hierarchy by allowing grouping basic states into super-states, which may at the same include complete state machines nested one into each other, without a limit on the levels in the hierarchy, and specifying conditions and transitions at a super-state level, reducing the complexity of the specification and improving the readability. When a super-state is left (there is a transition from that super-state to another one) all the states included in the super-state are disabled, and when the super-state is active again the state machines inside are enabled again. The nested state machines may either revert to a default start-up state or may continue in the state they were in when the super-state was left, which is called history. This behaviour is defined on the specification of the statechart. Some super-states may be active concurrently running in parallel, each of them and their nested state machines independent of each other, making them suitable to describe complex real-world systems, and the conditions to enable or disable them can be specified in an unambiguous manner. To support these features, statecharts also implement a better communication among the states than FSMs. Therefore, statecharts largely improve traditional state machines, allowing for more compact, expressive and modular diagrams, that can describe more complex behaviour than FSMs.

As well as FSMs, statecharts are very easy to understand, even for non-experts, as they are just diagrams to describe the behaviour of systems, separating the behavioural description from the component description. This

characteristic allows to test and modify the behaviour independently from the rest of the system. Statecharts and FSMs also make it easier to debug and find no-exit cases when compared to the source code implementation. Compared to FSMs, statecharts scale very well as they include concurrency and hierarchy. All of this makes them a very useful and complete representation of systems.

Therefore statecharts maintain all of the characteristics of FSMs, such as conditions, outputs, etc, with some important contributions that can be summarized as:

- Orthogonality: as opposed to classical FSMs, where only one state can active at a time, statecharts can have more than one state active concurrently. These are called *AND-states*, while the traditional approach are called *OR-states*. Orthogonality is very useful for describing subsystems.

- Depth: there is a hierarchy in the state structure, allowing for states or even complete FSMs or sub-statecharts to live inside other states, with transitions at every level, even in the states containing other states. In the nested structure the state containing other states is called *super-state*. Depth allows for great modularity, clustering, and ease of movement between levels of abstraction by zooming in or out. It is also possible to define entry and default states, and have *history* in the states, as explained in Section 4.3.3, that get active when there is a transition to the super-state containing them.

- Broadcast mechanism for communication between concurrent components, global triggers, conditions and actions.

One of the consequences of orthogonality is that, although each region should run almost independently of the concurrent ones, they can affect each other, for example if one event is raised by one of the regions, and that same event has an effect on a different region. An example of this is shown in Figure 4.2, where the state `counting` in the region `pedestrians` will trigger event `p` when left. At the same time this event will trigger the transition from `green` to `yellow` in region `cars`. It is also possible that an event triggers transitions in two different regions, such as if in the same figure the states `pedestrians` and `red` were active and event `c3` happened. In this case the event would trigger simultaneously a transition from `pedestrians` to `wait` in region `pedestrians` and from `red` to `green` in the region `cars`. In

this case the transitions occur *simultaneously*, which shows another feature of the *orthogonality* of statecharts: there is *synchronization*. If in a similar the state `red` is not active but `pedestrians` is, the same event `c3` will only trigger a transition in the region `pedestrians` but not in the region `cars`. This is called *independence*, and is another feature of *orthogonality*.

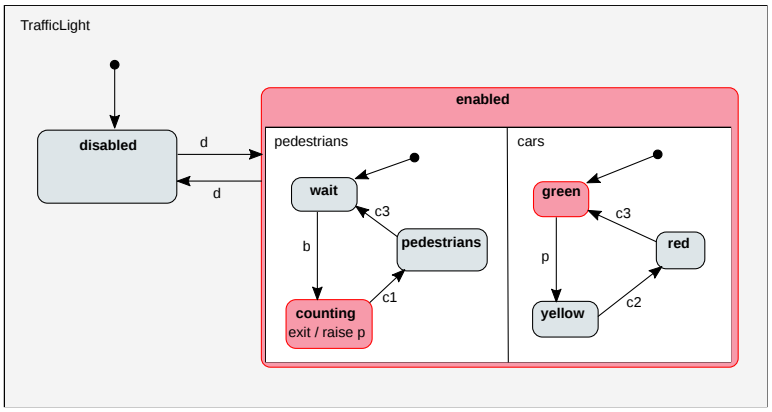

Figure 4.2: Statechart showing orthogonality.

4.2 Hardware synthesis of statecharts

Although the main use-case for our application is the implementation of an embedded EVR in the miniIOC from only a graphical description of an EVR represented as a statechart, our application is capable of creating synthesizable VHDL code for any system that can be described as a statechart, using this statechart as starting point. This automated methodology also keeps the chance of errors as low as possible.

The description of systems as statecharts and their automatic hardware synthesis has a number of advantages over the classical method of manually implementing those same systems by hand:

- The specification is done at a high-level, with little required training, thanks to the visual character of statecharts.
- Once the statechart has been implemented, future changes to that statechart are very easy, fast and do not require much extra effort from the engineer; this applies even to more complex changes such as

adding instances of sub-systems or porting functionality from other projects.

- Our application is hardware-independent, there is no need of taking the hardware characteristics into consideration when implementing a statechart, since it only describes the behaviour of the system.

Some tools for hardware synthesis were developed soon after the introduction of statecharts, although they were based in obsolete technology or has a limited scope. In 1989, Drusinsky and Harel [49] studied the implementation of depth, focusing on the activation and deactivation of super-states, the history and its challenges. That work focused on the implementation on PLAs[1], and was later analysed by Drusinksy and Yoresh [50], who stated its limitations. They also focused on PLAs and came to the conclusion that flattening the statechart into an equivalent FSM is inefficient, and proposed reducing the complexity of the transitions between super-states and coding the states in a more efficient way. [51] described some use cases of automated synthesis with a tool called I-Logix Express VHDL, the earliest automated tool mentioned in the bibliography. The tool has a reduced scope, since it does not support most of the statechart features, such as history. The same tool is used in [52] but focuses mainly on validation overlooking the details of the implementation, such as history. A latter product by I-Logix (Statemate) supported hardware synthesis but with some limitations. This software evolved later to become Rhapsody (now owned by IBM), but in the process it lost support for synthesis.

A number of works have been also published comparing VHDL versus statecharts synthesis. In [53] SPeeDCHART is used, and the authors point up that this tool is not complete and that a strong VHDL knowledge is still required. Another paper by the same authors [54] includes a comparison based on fuzzy control systems. However, the provided examples are actually implemented as FSMs, as they lack of the sophistication of real statecharts. A different work [55] proposes the mapping of statecharts using an ASIP[2], but the architecture of the ASIP is the main focus of the paper. The analysis of statecharts and VHDL implementation are done with a tool called *ROOM*, and are incomplete and barely discussed, lacking features such as history. This is not solved in another work by the same authors [56]. Nevertheless they mention some limitations of Statemate, stating that two

[1] Programmable Logic Array.
[2] Application Specific Instruction Processor.

AND-type super-states with n and m states each, are better implemented as a FSM with $n \times m$ states, essentially flattening the statechart in some degree. The implementation of FSMs with datapath is studied in [57], using again the SPeeDCHART tool also used in [53] and [54]. The authors notice that the tool can not implement concurrency successfully, since it flattens super-states into a single super-state with a large number of states and transitions. As in most of the works, history is not discussed.

More recently, a number of papers [58,59] have been published analysing the theoretical basis of automatic implementation of statecharts into Verilog HDL. They also introduce a statechart editor and a hardware mapping tool in [60] that also includes some examples. Nevertheless this implementation is not complete since it does not support history nor transitions between states at different levels.

A methodology is presented in [61] that converts a statechart specification into SystemC and VHDL, although the paper is mostly focused on SystemC and there are little details about the generation of VHDL code. The paper also focuses mainly on how to guarantee consistency between the SystemC and VHDL code behaviour when triggering events during the simulations. Also, as in many other works, history is not supported.

Finally, Mathworks has released an HDL coder for their commercial tool Stateflow [62] that would extend statechart processing by including an HDL generation back-end. On top of being a proprietary and commercial tool, we did not have the opportunity to test it, and we could not find any review that addresses how much of the statecharts functionality is actually supported.

In summary, a number of tools were proposed for statechart-based hardware implementation. However, they covered only a partial set of features and they are now discontinued. Meanwhile, software synthesis is present in many modern tools. In the remaining of this work we will describe one of them and how to extend it to produce synthesizable hardware.

4.2.1 Graphical representations of statecharts

Following Harel's formalism first publication, a number of graphical representation tools have been developed [63, 64], such as the previously men-

tioned Statemate. The earliest ones are just used to create graphical representations of statecharts, without any added value. Soon after that the tools improved and also represented the graphical representation as mathematical realisations, mainly using the UML notation [65], which allow not only the modelling but also the development, specification and verification in an automated way. Some of these can also translate the statecharts described in the UML [48] and do an implementation in some other programming language, such as C or Java. One of those tools is Yakindu Statechart Tools (Yakindu SCT) [66] based on Eclipse [67], which provides a graphical diagram editor where the user can design the statecharts. It also includes a statechart behavioural simulator and validator for the verification of the statechart, and code generators that can automatically translate the mathematical realisation of the statechart into Java, C and C++ source code, without needing any external library, and with a well-defined interface for an easy integration into any other code. Yakindu also has a custom code generator for exporting the statechart into other programming languages, although in that case the user needs to build the translator by himself or herself, with only a translator template being provided by Yakindu. Figure 4.1 was generated using Yakindu.

Custom code generators for Yakindu for VHDL code

There have been some efforts into generating VHDL code from the Yakindu representation of statecharts, being the main one [68]. These implementations are only partial since they lack important features, such as support for history, impose important limitations on the hierarchy or transitions between states, or they do not produce synthesizable code, generating only code snippets that still need to be integrated in a bigger project, and most of them are not published anywhere[3]. Our implementation does generate a synthesizable VHDL source code file ready to be used as a stand-alone in a real-world application.

4.2.2 Parsing and analysis of statecharts

Yakindu saves the mathematical realisation of the statechart in Extensible Markup Language (XML) that is both human- and machine- readable. The

[3]This assessment is based on the pieces of software found on-line. If a more complete implementation exists, neither the code or the description have been made public.

XML file has two parts: the first part describes the mathematical representation of the statechart in terms of its states, transitions, regions, conditions, etc, and the second part describes the graphical representation of the statechart diagram as it was drawn in Yakindu (basically size and location of the elements, but also colours, shapes, borders, etc). This graphical description of the statechart, as it was drawn, is of no interest for the automated synthesis tool, so we will ignore it for the rest of this work.

The statechart part of the XML is organized in a hierarchical way, starting with the highest object (the main region) which includes (is parent of) one or more states or entry points, called *vertices* in a generic way. At the same time each state can contain more regions and *outgoing transitions*; vertices can also have nested children, without any limitation on the number of nested levels or number or elements per level. Each element in the statechart is also given a set of attributes: always a unique *ID*, and, depending on the nature of the element, also a *type*, a *name* (which only needs to be unique in the local region or state), a *specification* (which includes the conditions, events and actions), a *target* (only for outgoing transitions, is the ID of the target state), *incoming transitions* (which are the IDs of outgoing transitions), and a *kind*. In this way, every transition is represented by two objects: the *outgoing transition* is an element on its own, a child of the source state, on the other end an *incoming transition* as an attribute of the end state.

The application that we have implemented is based on Xerces-C++ [69], a validating XML parser written in C++. Xerces-C++ includes a shared library that provides the functionality to parse, generate, manipulate and validate the XML documents with different APIs[4]. For the implementation of our application we have used the DOM[5] [70] API.

Our application parses the XML file one node at a time, aware of its exact place in the hierarchy in each step. For each node it keeps track of the list of parent, sibling and children nodes, while reading the list of attributes. With this information the tool creates the appropriate VHDL output code for the current node before moving to the next node in the structure. To complete the generation of the VHDL file our application parses the original file several times, once for each section of the VHDL file.

[4]Application Programming Interface.
[5]Document Object Model.

The output of our application is a synthesizable VHDL file that implements in a FPGA the same algorithm that was described in the original statechart modelled in Yakindu. Our application only expects one input, which is the XML file describing the statechart, as it is produced by Yakindu, to generate the output VHDL file. In the process our application can automatically detect the inputs and outputs of the statechart, creates the adequate signals, defines all the needed processes, and detects the transitions.

Although Yakindu includes a custom language code generator, the tool that we have developed does not make use of it, and instead it only uses Yakindu as the front-end to interface with the user and create the mathematical representation in XML code of the statechart designed by the user. Our application is actually the back-end that, after the XML file is created and without making use of the Yakindu custom language code generator, generates the VHDL code. This means that our application can be used with representations of any source, not just Yakindu-generated XML files, although it expects that the structure of the file, as well as the naming and separators, is exactly the same, so statecharts that have a source other than Yakindu should follow Yakindu's XML convention.

In its current configuration, the tool is developed for generating only VHDL code. It would be possible to extend it to generate code in other languages, but with an obvious focus on hardware description languages since those are the goal of the application; this is explained in Section 4.3.9.

4.2.3 Restrictions on the original statechart

In our current implementation some restrictions are assumed in order to ease the design of the application:

- The names of the elements do not include spaces and are not reserved keywords in VHDL, such as `begin`, `component`, `process`, `wait`, etc.
- Only events and internal conditions are accepted as transition triggers: external events, counters or integer variables having a specific value are accepted, but more general, "smart" conditions, such as *"after one second"*, valid in the Yakindu model, are not accepted. It is possible though to create a counter that triggers after an integer number of cycles of a clock, which would replicate the *"after one second"* condition.

This should be done when designing the statechart in Yakindu, and there is no need of knowing the internal structure or way of working of our application.

- With the exception of *regions*, most of the new features included in the UML implementation of statecharts but not in the original Harel's paper are not supported. This includes *exit points*, *choices*, *forks*, *joins*, etc.

- Only *shallow history* is supported: *shallow history* is defined as a *pseudo-state* that remembers the last active state inside the region that includes it. *Deep history*, which remembers all the latest states of a hierarchy of multiple nested states, is not supported. Instead of using a *deep history* node, a *shallow history* node can be included in all levels of the hierarchy to achieve the same behaviour, so this forces a different specification of the statechart but does not limit the functionality that can be implemented.

- All the transitions occur between states in the same region and level of the hierarchy.

Most of these restrictions are in place to rule out cases that are improbable or even impossible in our goal implementation. Others, such as the two last points in the previous list, does not inhibit the implementation of any system, although it forces to re-design the statechart. The very last point forces some non-trivial changes to the statechart, including some extra transitions, conditions and states in the statechart, which is not optimal, as simplicity and compact format are key features of statecharts, so we will explain how to do it:
Let us take a look at the statechart shown in Figure 4.3. The transition triggered by event `T` does not fulfil the last condition in the previous list. This transition can be replaced for another transition between the two states in the region of the statechart that, somewhere down in the hierarchy, include both the source and target states of the transition triggered by the event `T`, so the new transition goes from state `A` to state `D`. The new transition can be seen in the equivalent statechart in Figure 4.4 and has a condition, which is that it only takes place when the original source state `C` is active. Some additions to the new target state `D` are also needed: an extra state is needed, called `guard` in the example, and the *entry point* should point to this new state. In this example the new target state is the parent of the original target state, but if there were to be more levels in the hierarchy

between them, then all of them need an extra state, with the same features. This new state `guard` has two transitions: one transition to the original target state with the condition that the original source state was active; and a second transition to the default state pointed to originally by the *entry point*. If the *entry point* had *history*, it should be replaced with a simple *entry point* without *history*, and a variable that takes different values based on the state that the *history* needs to go back to. This variable is used in the transitions from the new state to the other states. The new state should be added in all levels of the hierarchy mentioned earlier.

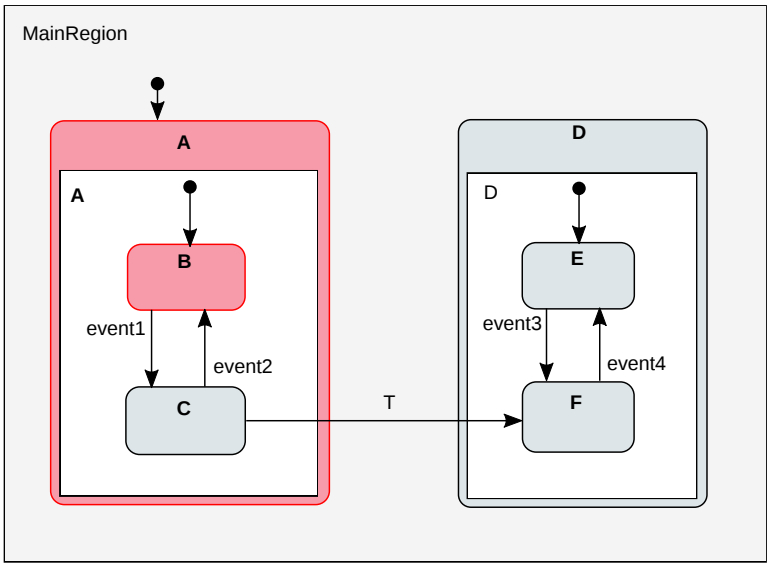

Figure 4.3: Statechart showing a forbidden transition.

The one problem that this method entails is that the transition that is replaced now takes one more clock cycle per level of the hierarchy between the original target state and the new target state, since there is one more intermediate state in each level of the hierarchy between the states.

4.3 Implementation strategy

The synthesis of FSMs is well understood [71,72], and a number of tools exist which are able to convert graphical representations into HDL code. However, statecharts introduce new features, such as super-states and history,

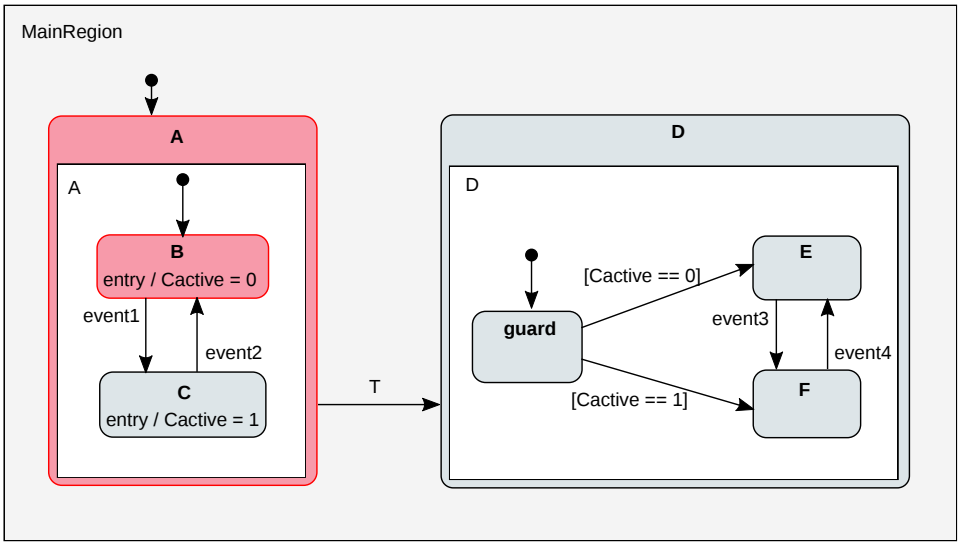

Figure 4.4: Statechart equivalent to the statechart shown in Figure 4.3 but without the forbidden transition.

that require new synthesis techniques. So far, synthesis has been addressed from a software point of view [73].

By analysing the XML description of the statecharts provided by Yakindu, a basic implementation of a super-state can be obtained using the same procedure as for FSMs, which consist of defining: (a) state encoding, (b) a state transition function, and (c) an output generation function. However, some characteristics of super-states may be more challenging to implement, such as:

- Orthogonality (or concurrency).
- Depth (or hierarchy).
- History.
- Distributed generation (super-states generating the same output or event).
- Entry, exit and event-driven actions inside a state, conditions, and actions on transitions.

4.3.1 Orthogonality

While traditional FSMs can only have one state active at a time, statecharts can have two or more states active concurrently. This can happen when the two states are in different levels of the hierarchy, as explained in Section 4.3.2, or in the same level (orthogonality). In the latter case, they are called *AND*-states. The key point about orthogonality is that, and following the UML implementations of statecharts, when the *AND*-super-state including the parallel or concurrent regions is active, the statechart is in *all* of those regions at the same time, and one state in each of the regions is always active. It is also said that the statechart is in the *orthogonal product* of those active states. In fact, each region implements a FSM or even complete sub-statechart, and all of them are active at the same time. For example, let us consider the statechart shown in Figure 4.1. Inside the `active` super-state there are two parallel regions, named `send` and `receive`, and each of them has an active state: the `send` region is in the `wait` state and the `receive` region is in the `read` state. The result of this is that the `active` super-state, and the statechart itself, is in the *orthogonal product* of `wait` and `read`. Orthogonality also means that, when there is a transition to the `active` state, the transition is also to the combination of the `idle` and `read` states, as indicated by the entry nodes. When the event `sleep` triggers, and the `active` super-state is disabled, so are whatever state was active in the `send` and `receive` regions.

Our application implements each region as a different VHDL process that runs concurrently with the rest of the regions (processes). These processes implement the transitions between states with *case...when...* structures. This fulfils the *orthogonality* feature:

- VHDL processes are inherently concurrent, and thus they run in parallel without forced interaction between them, as is expected from different regions inside an *AND*-super-state. Interaction between different regions can still be achieved using events and variables.

- Processes can have the same signals in their activation lists, resulting in *synchronization* of the processes, assuming also that all the transitions are triggered with the same clock in the *synchronization process*, that is explained later in Section 4.3.6.

- The *case...when...* structures implement the *independence* feature: transitions only take place the correct state is activated.

Listing 4.1 shows an extract of the XML file of the model that describes the two parallel regions, and Listing 4.2 shows an extract of the VHDL code generated by our application that implements these two regions as processes.

```
<vertices xsi:type="sgraph:State" xmi:id="..." name="active"
    incomingTransitions="...">
  <outgoingTransitions xmi:id="..." specification="sleep" target="..."
      />
  <regions xmi:id="..." name="send">
    [...]
  </regions>
  <regions xmi:id="..." name="receive">
    [...]
  </regions>
</vertices>
```

Listing 4.1: Extract of the XML file from Figure 4.1 that describes the two parallel regions (edited for clarity).

```
sendFSM:process(sleep, wake, send, resume, background,
    sendCurrentState)
begin
  case sendCurrentState is
  [...]
  end case;
end process;

receiveFSM:process(sleep, wake, send, resume, background,
    receiveCurrentState)
begin
  case receiveCurrentState is
  [...]
  end case;
end process;
```

Listing 4.2: Extract of the output VHDL file implementing the two regions described in Listing 4.1 (edited for clarity).

4.3.2 Depth

As opposed to FSMs, statecharts can have different levels of hierarchy with states (or even complete FSMs) living inside other states, giving the statechart a sense of depth. In this case, when a super-state is left, all the states underneath it should be disabled, and when the super-state is entered again, the FSM inside it starts or continues its operation. This is one of the key characteristics of statecharts when compared to FSMs: there is a *clustering* of states into super-states, achieving a reduced number of transitions. One example is shown in Figure 4.5, where the two regions implement the

same functionality, but while the left one has three transitions, excluding the entry transitions, the region on the right only has two transitions. In this example we have created a new super-state `D` that clusters states `A` and `B`, and replaced the two transitions triggered by the event `event1` for only one transition triggered by the same event. While this is a simple example, in more complex statecharts this allows keeping the number of states and transitions low, which is one of they goals of statecharts.

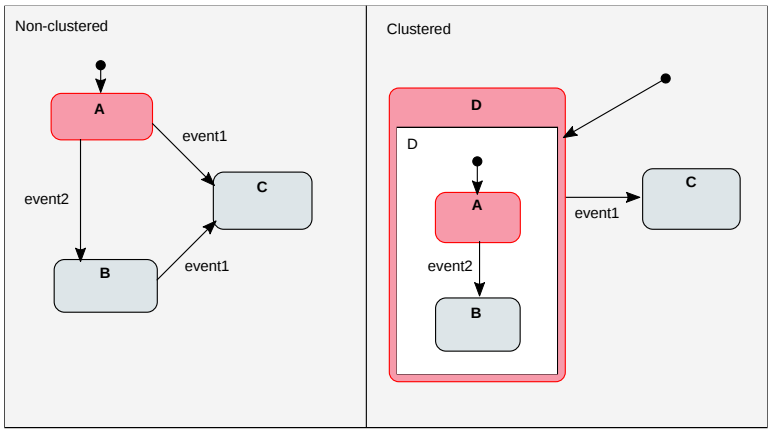

Figure 4.5: Statechart showing *clustering*.

Another characteristic that comes with *depth*, is that it is mandatory to specify default states or *entry nodes* for each level in the hierarchy. The *entry nodes* point to the state in each region that will get active when there is a transition to the parent super-state. There are two kinds of *entry nodes*: *entry nodes* pointing to default states and *entry nodes* with *history*, which are discussed in Section 4.3.3.

Depth can be implemented in two ways:

- Each region is automatically disabled when the super-state containing it is disabled, and it is also re-established automatically when the parent super-state is enabled. This means that each region in the hierarchy needs to be constantly monitoring if the parent state is active or not and disabled its own states accordingly, and then it will be cascaded down: the children regions will also be monitoring the containing state and get disabled when the parent state is, and so on.
- Each region does not monitor if the parent states are active or not,

but instead it is aware of the transitions to and from the parents and acts based directly on those. In this case the implementation of the state does not need to add new functionality, but instead it just adds new events to the list of events that it was already monitoring, and the action (already implemented) of getting disabled/enabled triggered by these new events.

Although in some cases the first option could be more suitable, it is obvious that the second option is easier to implement, since the functionality already exists. There is one more reason that also makes the second option a better choice: the tool may be improved by removing the requirement of transitions occurring between states in the same region and level of the hierarchy, allowing for a transition from one super-state to a state inside another *OR*-super-state. In this case it would be the sub-state being enabled that also forces the super-state to enable, and not the other way around. There are two ways of doing this: by monitoring the transitions to and from all the states in the hierarchy, or by monitoring when all states get enabled or disabled, and cascading that both up and down in the hierarchy. While monitoring transitions is already implemented, monitoring all the states changes and cascading is not implemented and would be more resource expensive. Also in the same case when we remove the same offending condition, it would be needed to override the *history*, since the active state should be the one pointed to by the transition. This means that it is not enough with monitoring the change of the states, since knowing what transition triggered the change is also important.

Listing 4.3 shows the output of our application for the `wait` region in Listing 4.4.

```
backgroundFSM:process(sleep, wake, send, resume, background,
    backgroundCurrentState)
waitFSM:process(sleep, wake, send, resume, background,
    waitCurrentState)
begin
  case waitCurrentState is
  when idle =>
    if wake = '1' then
      waitHistReg <= 0;
      waitNextState <= waitEntry;
    else
      if background = '1' then
        waitNextState <= background;
      end if;
    end if;
```

```
  when background =>
    if wake = '1' then
      waitHistReg <= 1;
      waitNextState <= waitEntry;
    else
      if background = '1' then
        waitNextState <= idle;
      end if;
    end if;
  when waitEntry =>
    if sleep = '1' then
      case waitHistReg is
      when 0 =>
        waitNextState <= idle;
      when 1 =>
        waitNextState <= background;
      end case;
    end if;
  end case;
end process;
```

Listing 4.3: Extract of the output VHDL file implementing the `wait` region (edited for clarity).

```
<vertices xsi:type="sgraph:State" xmi:id="..." name="wait"
    incomingTransitions="... ...">
  <outgoingTransitions xmi:id="..." specification="wake" target="..."/
      >
  <regions xmi:id="..." name="wait">
    <vertices xsi:type="sgraph:State" xmi:id="..." name="idle"
        incomingTransitions="... ...">
      <outgoingTransitions xmi:id="..." specification="background"
          target="..."/>
    </vertices>
    <vertices xsi:type="sgraph:State" xmi:id="..." name="background"
        incomingTransitions="...">
      [...]
    </vertices>
    <vertices xsi:type="sgraph:Entry" xmi:id="..." kind="
        SHALLOW_HISTORY">
      <outgoingTransitions xmi:id="..." specification="" target="..."
          />
    </vertices>
  </regions>
</vertices>
end process;
```

Listing 4.4: Extract of the Statechart file implementing the `wait` region (edited for clarity).

The hierarchy can have any number of levels, so a sub-state in a super-state can at the same time be a super-state. Our application uses recursive functions to go through all the levels of the hierarchy to complete the stat-

echart model, as is explained later in this same chapter.

4.3.3 History

A super-state has *history* if it can remember its present state and return to it later after being disabled for a while. A super-state without *history*, however, will always resume its activity starting at its initial default state. Implementing this behaviour can basically be achieved in two manners, compared in Figure 4.6:

- In the first way, a wait state is added, and an enumerated variable of the current state is kept on a register that is connected to that wait state. A disabled super-state will be running in its wait state until a resume event is triggered, which can be directly targeted to the super-state or to the super-states higher in the hierarchy, as explained in Section 4.3.2. At that moment the wait state will trigger a transition to the state the region was in before the super-state was left, according to the value stored in the register.
- Alternatively, it may be simpler to add a wait state attached to each normal state and transit to it when the super-state is disabled. When the super-state gets active again, the wait state that was active transitions to its attached state, which was the active state before the first transition was triggered.

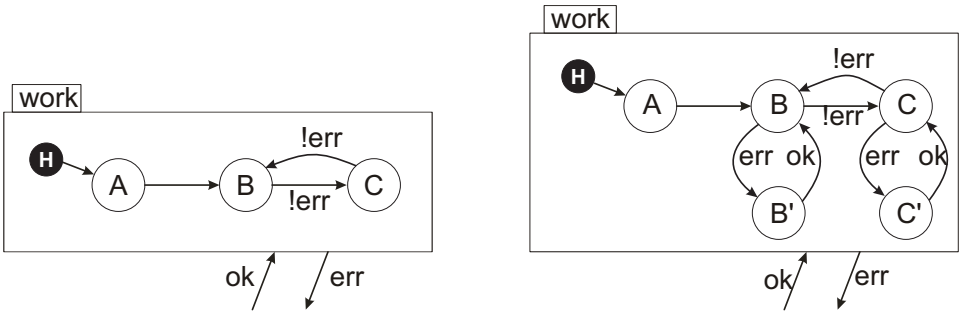

Figure 4.6: (a). super-state with history. (b). Proposed implementation using wait states.

The convenience of each solution should be assessed for each particular case but, in our current application, only the former is implemented,

which can be seen in Listing 4.3. In this example the wait state is named `waitEntry` and it stores the state in the `waitHistReg` register. The wait state is also used as the entry node, but its initialization is not shown in the listing.

There is another reason for choosing the first solution instead of the second, and it is that in that case it would be possible to clear the history, something that is introduced by Harel but that apparently is not possible in Yakindu. Clearing the history is a special action that resets the history *entry point* so that it points back again to the default state, instead of to the previous active state.

4.3.4 Distributed generation

Ideally, each super-state generates a sub-set of outputs and signals different to other super-states. However, this is not always true, either because the statechart is not neatly designed or because using different super-states actually helps to organize the statechart. Hence, the implementation of each super-state may conflict with others. A preliminary parsing of the statechart description finds which outputs and signals are produced by different super-states, and asserts that only one of them is active at the same time. Next, output and signals are renamed by appending a suffix that relates each signal to the super-state in which is produced. Finally, the global signals are obtained by reduction as: $globalSignal = (signal\_ss1\ and\ active\_ss1)\ or\ (signal\_ss2\ and\ active\_ss2)\ or....$ This is depicted in Figure 4.7.

4.3.5 Actions and conditions

Entry and exit actions are not unique to statecharts, since FSMs can also model them as part of their inputs and outputs. But, in statecharts, they are expanded by allowing actions to happen when some conditions are met at any moment during the active phase of a state. For example, a state may increase a counter when an event is raised anywhere in the statechart, but the counter would not increase if the specific state is not active. In this case the statechart models the counter-increase action as being performed by the active state. Conditions can also apply to any transition, action, event, etc. Figure 4.8 shows a simple statechart implementing some of this cases.

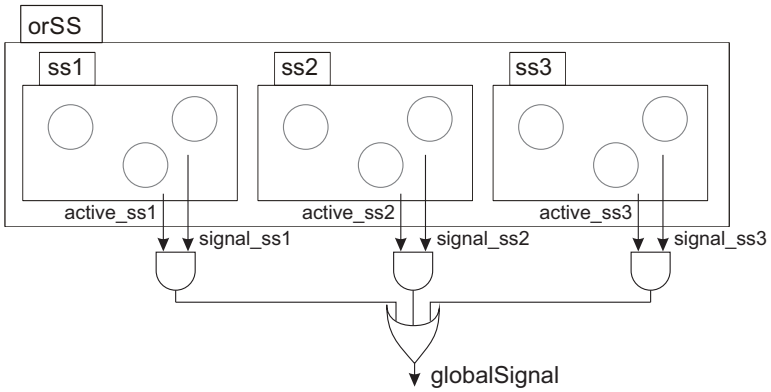

Figure 4.7: Distributed generation of a global signal.

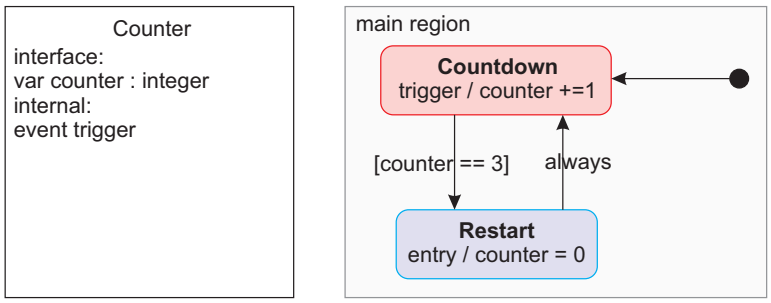

Figure 4.8: Simple statechart with actions and conditions.

4.3.6 Implementations steps

The application that we have developed requires two arguments: an input XML file describing the statechart and a name for the output VHDL file. Then the tool applies a number of steps to generate the output VHDL file. The application is built as a main function that only gets the name of the input file, the output file and calls the driving function, which does most of the work, as explained in the following subsections.

Initialization of the parser

The input file is tested to see that it can be opened correctly. In that case and if it includes a valid XML architecture, the parser is set, it starts the parsing of the file by creating a DOM document and creates an element for the statechart. A DOM node is also created, which is first set to the first child of the statechart element, which is a node pointing to the main statechart. At the same time the VHDL file is created and the used library is defined.

Entity declaration

The first step is the declaration in the VHDL of the entity with the name of the statechart. An input clock is included by default, and the *specification* of the statechart node is parsed to look for the variables and events in the interface list that complete the list of inputs and outputs of the entity.

Type and signal definition

Then the first three parsings of the XML file take place. The goal of these firsts parsings is defining the list of VHDL types and signals. These are basically a list of the regions in the statechart, states and history nodes:

- A VHDL type is defined for each region, which can have the values of the states included in that region.
  The `GetStates` recursive function parses the XML file node by node starting at the top statechart node. For each node it checks if the first child is a region. In that case it creates a new type for the region, `LocalRegion`, and looks for grandchildren vertices. The grandchildren vertices, which are children of the region, are states and entry nodes, and are listed as the possible values of the region/type. Then the recursive function is called again from within each of the found states, until the deepest point in the hierarchy has been reached. At that moment the current node pointer is set to the next sibling of the current node, and the recursive function is called again. When the last sibling has been parsed, the pointer moves up one level in the hierarchy and the same process takes place again for the next sibling. This continues until all the statechart has been parsed and the complete list of types has been defined. All the previous is done while keeping track of the current point in the hierarchy.

- Two signals are defined for each region: a `LocalRegionCurrentState` and a `LocalRegionNextState`, which are of the type corresponding to the region under consideration.
  A very similar approach to the `GetStates` recursive function is used with the `PrintStatesSignals` recursive function to define the two `LocalRegionCurrentState` and `LocalRegionNextState` signals for each region. The application will use this signals for the transitions from one state to another. The default state is also define here, according to the *entry node*.

- One signal, of type `integer` and called `LocalRegionHistoryRegister`, is defined for each history node in the statechart.
  When a region containing a history node is left, this signal will save the active state so that it is possible to come back. This is done with the `GetHistoryRegisters` recursive function, which works in a similar way to `GetStates` and `PrintStatesSignals`.

Since regions may have the same name as other regions in other states but the types need to have a unique name, the name of each type is defined by concatenating the name of all the nodes from the top level down to the region that the type represents.

In these parsings also the list of outputs with several sources and their parent states is created. Also the partial output signals and their related state-active signals explained in Section 4.3.4 are defined.

Structural description

Before tackling the transitions, the XML file is parsed once to get a structural description of the statechart. Since Yakindu identifies each element (including states, transitions, etc) with an ID instead of the name, the `CreateStateIDList` function is needed to create a list of pairs `state name - ID`, which also includes the entry nodes. This list will be mainly used for identifying the correct target state name in each transition, since the state is identified in the transition specification by its ID instead of its name. This list is used in some other cases too. The application could still work using the IDs instead of the actual state names, since IDs are unique and define perfectly the states. The reason for creating this list is that the IDs loose any meaning, since they are almost arbitrary strings of characters, such as Yakindu defines them. We have decided to work with the names using the

`state name - ID` list instead.
The `CreateStateIDList` is also recursive, and parses the statechart file from beginning at the *top* level to the end, going node by node, first checking the current node, then visiting the first children node and repeating the process. When all the children nodes and nested nodes are checked the parser moves to the parent node and the process runs again for the siblings, if any, until all the statechart has been parsed.

Another function, `CreateOutTransIDList`, does exactly the same but for a list of outgoing transitions.

Transitions

Next the transition logic begins. This is the most complex step, where the transitions are coded, the states are activated and deactivated and actions take place, and where all of the considerations explained before about *orthogonality*, *depth*, *history* and so on express themselves. Most of the next remarks can be seen applied in Listing 4.3. As mentioned in Section 4.3.1, basically each region is implemented as a different VHDL process that runs concurrently with the rest of the regions (processes). These processes implement the transitions between different states with *case... when...* structures. In these structures, the *case* is the `LocalRegionCurrentState`, which can have the values defined in the function `GetStates`, this is the names of the states and entry nodes in that `LocalRegion` type, as explained before. The names of the states are the *when...* values.

In each of the states, or *when* cases, a series of *if... then... else if... then...* structures define the transitions between states: for each event that can cause a transition from that state to another one, an *if* statement checks if the event is `true`, and *then* it sets the `LocalRegionNextState` signal to the state pointed by that transition. Also in these processes is where all the actions take place. Actions can be divided into two groups: the first group includes actions that are attached to the transitions themselves or are part of the *entry* to the target state or *exit* from the source state. In this case the action is instantaneous (for example, raise an event) and is implemented inside the *if... then... else if... then...* structures. The second group are actions that take place or need to be evaluated during all the time that the state is active, like in a Moore machine. In this case the action is implemented inside each of the *when...* states but outside of the transition

if... then... else if... then... structures. As explained in the Section 4.2.3, the UML implementation of statecharts is inherently asynchronous, but the implementation in FPGAs makes them synchronous to some signal, usually a clock, so some limitations are imposed in the possible actions. It is important to note that our tool defines a clock for synchronizing the transitions, and this master clock can not be used in the conditions; if a clock is needed for some periodic effect of the actions, a different clock than the master synchronization clock needs to be defined. This should be done in the interface list in Yakindu, there is no need to do it manually in the VHDL code.
One specific type of actions is output generation. When those actions are implemented, they are compared with the list of outputs with several sources, and if they are in the list the action is implemented to the partial signal, not the global one. Also the state-active signals are driven accordingly when setting the `LocalRegionNextState` signals.

The history pseudo-states are a little bit different since they only have one goal: keep the return-to enumerator and perform the transition when it is needed. For this reason the history pseudo-state *when...* case has as many *if...* statements as transitions can occur back to the region containing the history node, and each of these *if...* has only one *case...when...* structure, where the *case...* is the `LocalRegionHistoryRegister` and each of the *when...* are the enumerated integers, that trigger a transition of the `LocalRegionNextState` signal to the state saved in the history register, as marked by the enumerated value.

The main function that prints the processes as explained before and that implements all the transitions is `PrintFSMProcess`. Before analysing it, we need to mention some other functions that are called by this function, are important and get called several times:

- The `GetNameFromID` function returns the name of a node from its ID, thanks to the list created by the `CreateStateIDList` function. This is important since, as explained before, we use the names of the states to identify them, while Yakindu uses the ID. This function is used for example to identify the target of outgoing transitions.

- The `GetSpecificationFromID` function returns the *specification* of, for example, an outgoing transition. The *specification* is where transitions define the events that trigger them. They can also include actions.

- The `PrintActions` function reads the *specification* of states and transitions and prints the action using the VHDL syntax.

The `PrintFSMProcess` recursive function prints iteratively a VHDL process for each pseudo-state-machine in the statechart, and thus is the function that takes most of the load of the application. It parses the XML one more time to print all of the VHDL processes, one for each region. We have explained before how the VHDL implementation of the regions looks like, so now we will focus on how the `PrintFSMProcess` function works to get to the expected output. It starts by, for each region, printing the process name and activation list, which includes the events that may cause a transition and also the `LocalRegionCurrentState` signal. Then it parses the children of the region one by one, which are the states and entry points. For the *history* nodes it raises a flag that will be used later. If there is no history, the function prints the appropriate *when...* statement for the state or entry node. If the children under consideration is a state, it checks if it has a *specification*, and in that case it prints the specified actions. The function also looks for outgoing transitions from the state or entry, and prints the `LocalRegionNextState` for the transitions, as well as the specifications. If the history flag has been raised, then the function works in a slightly different way, since it also needs to save the current state to the `LocalRegionHistoryRegister` when there is an outgoing transition from the parent state, and also print the *when...* case of the wait pseudo-state, which returns to the correct state when the region is activated again. The function also parses upwards in the hierarchy, looking for transitions out of the higher levels.

Synchronization process

The VHDL code is finished with a synchronization process defined by the `PrintSyncProcess` recursive function that, synchronously to the main clock and starting at the top of the statechart and for each region, updates the `LocalRegionCurrentState` to be the `LocalRegionNextState` defined by the transitions in `PrintFSMProcess`. This process basically triggers the transitions themselves that have been defined by the `PrintFSMProcess`. As explained in Section 4.3.1, this is necessary for achieving *synchronization*.

4.3.7 Example

As an example of the code generated by our tool we will use the statechart shown in Figure 4.8. Listing 4.5 shows the XML model that Yakindu produces for that statechart, and Listing 4.6 shows the VHDL code generated by our application from the XML description.

```
<sgraph:Statechart xmi:id="_nCEd8JlXEeepsse2VK4HgQ" specification="
    interface:&#xA;var counter : integer&#xA;internal:&#xA;event
    trigger" name="Counter">
    <regions xmi:id="_nCHhQplXEeepsse2VK4HgQ" name="main region">
      <vertices xsi:type="sgraph:Entry" xmi:id="
          _nCR5UZlXEeepsse2VK4HgQ">
        <outgoingTransitions xmi:id="_nCWx0ZlXEeepsse2VK4HgQ" target="
            _nCTugplXEeepsse2VK4HgQ"/>
      </vertices>
      <vertices xsi:type="sgraph:State" xmi:id="
          _nCTugplXEeepsse2VK4HgQ" specification="trigger / counter +=
           1" name="Countdown" incomingTransitions="
          _nCWx0ZlXEeepsse2VK4HgQ _AaEoAJlYEeepsse2VK4HgQ">
        <outgoingTransitions xmi:id="_8bZzAJlXEeepsse2VK4HgQ"
            specification="[counter == 3]" target="
            _wHHW4JlXEeepsse2VK4HgQ"/>
      </vertices>
      <vertices xsi:type="sgraph:State" xmi:id="
          _wHHW4JlXEeepsse2VK4HgQ" specification="entry / counter = 0"
           name="Restart" incomingTransitions="_8bZzAJlXEeepsse2VK4HgQ
          ">
        <outgoingTransitions xmi:id="_AaEoAJlYEeepsse2VK4HgQ"
            specification="always" target="_nCTugplXEeepsse2VK4HgQ"/>
      </vertices>
    </regions>
  </sgraph:Statechart>
```

Listing 4.5: Simple example of a statechart.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter is
  Port (clk : in STD_LOGIC;
        trigger : in STD_LOGIC);
end Counter;

architecture rtl of Counter is

type CounterStates is (CounterEntry, Countdown, Restart);
signal CounterCurrentState, CounterNextState : CounterStates :=
    CounterEntry;
signal counter : integer := 0;
```

```
begin

CounterFSM:process(trigger, CounterCurrentState)
begin
  case CounterCurrentState is
  when CounterEntry =>
    CounterNextState <= Countdown;
  when Countdown =>
    if trigger = '1' then
      counter <= counter + 1;
    end if;
    if counter = 3 then
      CounterNextState <= Restart;
    end if;
  when Restart =>
    counter <= 0;
    CounterNextState <= Countdown;
  end case;
end process;


sync:process(clk)
begin
  if rising_edge(clk) then
    CounterCurrentState <= CounterNextState;
  end if;
end process;

end rtl;
```

Listing 4.6: VHDL code output of applying our application to the Statechart shown in Listing 4.5.

4.3.8 Evaluation

The code generated by our application is, in general, more verbose than the equivalent VHDL code written by hand by a firmware engineer, having about twice the number of lines. Regarding the FPGA resource utilisation there is, however, little difference between the implemented and synthesized code from our application and the handwritten one. Table 4.1 shows the resource utilisation by both codes for the statechart shown in Figure 4.1, while Table 4.2 shows the data for Harel's digital watch shown in Figure 4.9. In all cases the code is implemented and synthesized for Xilinx Kintex-7 FPGAs [29] using Xilinx Vivado.

In general the number of lines of the code and the amount of resources used is proportional to the complexity of the statechart (both for the implementation using our tool and the handwritten code), while the delay does

	Hand-coded	Automatic tool
Lines of code	148	232
Logic blocks	24	26
Flip-flops	43	43
Delay (ns)	1.29	1.33

Table 4.1: Comparison of the resource utilisation for the statechart in Figure 4.1.

	Hand-coded	Automatic tool
Lines of code	569	1483
Logic blocks	131	169
Flip-flops	140	156
Delay (ns)	1.59	1.61

Table 4.2: Comparison of the resource utilisation for the statechart in Figure 4.9.

not increase in the same way, since it is only affected by the most complex super-state. This means that aggregating more AND-super-states to a given statechart increases the number of lines and resources used but does not increase the delay, so even large statecharts can be implemented into fast circuits.

4.3.9 Extension of the application to other languages

The application that we have developed could be extended to generate code in other hardware description languages, not just VHDL, reusing most of the recursive functions that have been already developed. Two main changes are necessary, which would need to be applied for each new language supported:

- The first and obvious step would be to set the application to print to the output file the code lines using the selected language syntax and semantics. At the moment this process would be quite slow and tedious since the lines that write to the output file are hard-coded inside the body of the functions of the application, although a future enhancement of the tool would be to extract the hard-coded printed lines to a configuration file where the actual output of each function can be easily adapted to different languages.

- The second step would be to configure the driving function to call the other functions in the correct order to generate a synthesizable output in the selected language. This basically defines how the output file is structured in terms of definitions, processes, etc, which are very language-dependant.

4.4 Microprogrammed implementation

In the previous sections, a tool that implements statecharts as VHDL code ready to be synthesized and deployed was presented. This was done with the intention of easing and reducing the work needed to implement and deploy a new control configuration in FPGAs while keeping the chance of human-induced errors as low as possible. It would be possible to go one step further by taking the logical synthesis out of the process. This is be possible by using microprogrammed architectures [45,46] that implement statecharts with a high degree of upgradability.

These architectures would add an important overhead to the implementation process but would provide some relevant improvements, applying to both ASICs[6] and FPGAs, such as:

- The changes would not need to be re-synthesized, allowing for a quickly deployment.

- Maintenance of the control system would be easier, since firmware updates are not dependant on the version of the synthesis software. Common problems such as expiring licenses, devices that reach their end-of-support cycle and software that is being constantly evolving without final versions are not so critical any more for a facility that will run for decades.

- In mission critical applications, is quite common that only a few devices are certified, due to the huge cost of certifying more modern architectures. As an example, Intel's 80286 processors are still in use in many commercial aircrafts. Therefore, being able to update the configuration of an ageing device without depending on the manufacturer updating the synthesis software may be of great interest.

[6] Application-Specific Integrated Circuit.

4.4.1 Microprogramming

Microprogramming was widely used some decades ago for microprocessor design when computer aided design was not as developed and efficient as it is today. Design errors were quite common and often hidden in corner cases that would not be detected until the microprocessor was already being produced and available in the market. *Microprograms* make it possible to correct those mistakes by loading new control configurations to an already commissioned system to update it, even in-field. For our work-case, statecharts may be implemented as a microprogram by mapping super-states as concurrent processes. This is the main advantage of microprogramming, as it allows for even greater flexibility than the previous approaches to control.

Originally microprogramming was developed for FSMs, although it can be applied to statecharts as well. The main idea behind microprogramming is replacing the default selection mechanism of the next state in a FSM with a programmable memory, called the *control store*, that is used to select the next state based on a series of conditions. Some of these conditions are embedded in the *microinstructions* in the control store, that also hold information about what actions to perform in the current state, while other conditions are dependant on the variables of the FSM or statechart, and the value of the next-state mechanism itself. When the microinstruction is processed, it evaluates all of the conditions and gets a new microinstruction, which may represent a transition in the statechart or FSM. Super-states in statecharts may be mapped as concurrent processes in the microprogram.

Microprograms have some important advantages:

- Hierarchy can be easily implemented in microprograms. Transitions leaving a parent super-state can be implemented as conditions that are re-used in all of the children of that state, in all the lower levels of the hierarchy.

- Microprograms also deal very well with history, since the current value of the control store can be saved in a separate register while running a different part of the statechart and later go back to the same point.

- More importantly, and as we have mentioned before, microprograms are very flexible and easy to update, since after the microprogram machine has been designed, synthesized and loaded, simply updating the control store memory is enough to load a new microprogram and

all of its new microinstructions. This means that statechart implemented by the microprogram can be very easily updated, as it is only implemented in the list of microinstructions in the control store.

4.4.2 Mapping a statechart into a microprogram

In Section 4.2 we have mentioned a number of papers that have been published on statechart synthesis. However it looks like microprogrammed implementations of statecharts have never been proposed. In this section we present a number of challenges that need to be taken into account when implementing statecharts with microprograms.

Orthogonality and depth

One of the main challenges of statecharts is how to support concurrency, implemented as AND-states that communicate among them. The microprogrammed architecture addresses this as a number of processing elements running in parallel. Each element is implemented as an independent microprogram that runs one AND-state. The implementation of the microprograms in micromemories is addressed in detail in Section 4.4.3.

In the original paper by Harel, a digital watch is proposed as example, which we recreate in Figure 4.9. It will be used in the remaining of this chapter as it is complex enough to illustrate all the implementation aspects. In it, the `main` super-state is managed concurrently with the `alarm 1 - status`, `alarm 2 - status`, `chime - status`, `light` and `power` super-states, forming an *AND*-super-state. At the same time this *AND*-super-state is an OR-state to the `dead` state, since they can only run alternatively. The hierarchy in the *AND*-super-state is quite deep, including mainly *OR*-super-states, but new AND-states are used in the pairs `regular` and `beep-test` inside `displays` and `display` and `run` inside `stopwatch`. History is also used in several super-states, some of them within other super-states that also use history themselves.

The implementation of OR-states is simple, since the transition from one state to another is similar to jumping to another process in a sequential piece of software. Transitioning to an *AND*-super-state, on the other hand, involves creating more processes, since the AND-state will have a number of processes active concurrently inside it. The solution to this is to reserve some

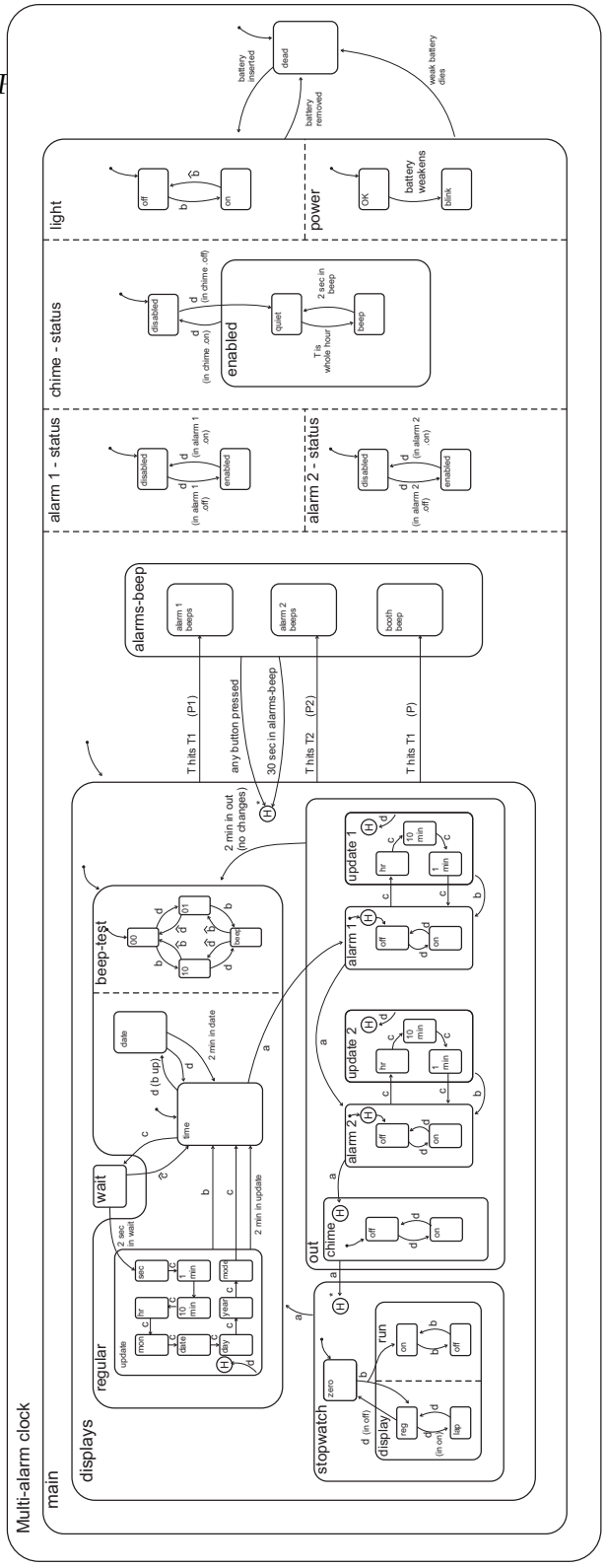

Figure 4.9: Recreation of Harel's example of a statechart to control a digital watch.

computing resources that are idle while they wait until the *AND*-super-state is activated. When activated the reserved resources are used to implement the children states.

The easy way of doing this is by implementing microinstructions that are able to send a message to another micromemory, on top of the ability to transition to a different microinstruction in a different super-state. This message would trigger the activation of the micromemory, which would load a specific microinstruction starting the local sub-statechart. The downside of this approach is that it makes the microinstructions very verbose, so a different implementation was chosen. In this alternative implementation inner *AND*-super-states are implemented as a number of ghost microprograms that are run idle microinstructions until a transition occurs that activates the *AND*-super-state. When the *AND*-super-state is activated it runs regular, meaningful microinstructions instead of the idle ones. Although this implementation looks very inefficient, usually ghost microprograms include significant parts of useful code.

Let us look at the left side of Figure 4.9, specifically the `displays` super-state, in `main`. It is possible to implement the `displays` super-state as two microprograms, as shown in Figure 4.10, that we named `main` and `ghost - main`. When the statechart starts the `main` microprogram does it in `time` and `ghost - main` in `beep-test`. When button `a` is pressed the state `time` transitions to a sub-state inside the `out` super-state, with the consequence that the `beep-test` super-state is disabled. In our microprograms this translates to `main` running fully functional microinstructions, while `ghost - main` only runs state transitions that do not update variables or produce outputs.

When the active state is `zero` inside `stopwatch`, both the `main` and `ghost - main` microprograms will be running the `zero` microinstruction, although in the case of the ghost microprogram the microinstruction will be idle. If button `b` is pressed in this moment, the main microprogram will transition to the super-state `display`, while the ghost microprogram will transition to the super-state `run`, both of them running in parallel and both of them active and functional. If button `a` is then pressed, the main microprogram will transition to the super-state `regular`, specifically to `time`, while the ghost microprogram will transition to the super-state `beep-test`, which is also the original or default situation.

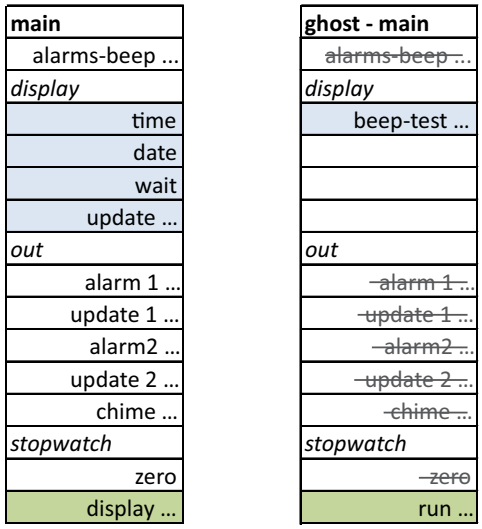

Figure 4.10: `Main` and `ghost - main` microprograms that implement the `displays` super-state. The microinstructions in the ghost microprogram replicate all the transitions taken by the main one, but they only perform actions when `ghost - main` runs the microinstructions that implement two specific AND-states: `beep-taste` and `run`. All other microinstructions are idle microinstructions and are struck-through.

History

History is one of the main challenges when implementing automatic synthesis of statecharts, as can be deduced from the fact that most papers, such as [52,56,61], do not consider implementing it. It is also observed that some restrictions apply in all the cases that we have checked. The main challenge is actually implementing history inside nested states in a correct way. Looking at Harel's example, let us consider again the `displays` super-state, which has history when returning from the `alarms-beep` super-state, as is declared by the `H` within the circle. In this case, the last state within `displays` that was active when the transition to `alarms-beep` happened must resume its operation.

In our implementation, however, history is only kept at the lowest level super-state for simplicity. This means that only shallow history is supported, and it is implemented at the lowest super-state level. Each super-state has a default re-entry state, but if the super-state implements history it will change the default re-entry state to point to the latest running state, while super-states that do not implement history will always use the default state. Applying this to Figure 4.9 means that if it is running in any of the specific super-states inside `displays`, and it transitions to `alarms-beep` and from there to `displays` again, it will always return to a pre-defined state. Let us say the pre-defined states are `time` and `beep-test` for example, although `chime` could have been the active state. However, if the active state was `1 min` in `update2`, whenever the statechart goes back to `update2`, it will also go back to `1 min` and not `hr`, because history works at that level. Although this imposes restrictions on the implemented statechart, history is still usable in a lot of cases, with possible workarounds in most situations that engineers should identify.

Microinstruction format

Orthogonality, depth and history are the main challenges to solve when mapping a statechart into a microprogram. So far we have addressed how to implement them in a microprogram, and now we will propose a format for the microinstructions that implements them and study the restrictions that it forces to the microprogram.

The microinstructions that we propose have two parts, which are shown

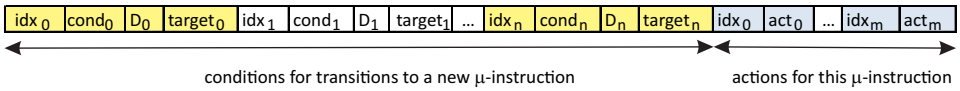

Figure 4.11: Proposed microinstruction format showing $n+1$ conditions and $m+1$ actions. The length of most fields depends on the number of allowed counters, inputs and outputs; or the maximum number of microinstructions in an *AND*-super-state.

in Figure 4.11. The first one is a condition evaluation part to decide the next microinstruction to run. The conditions are evaluated one by one in the order that they are included in the microinstruction, and when one condition is fulfilled, the remaining ones are ignored. Conditions use inputs and internal variables as parameters, and are implemented as counters, as explained in Section 4.4.3. Each condition has a chaining bit (D_i) used to apply the logical expressions *and* and *or* to multiple conditions at the same time. *And-ing* is applied to a condition by setting its chaining bit to *on*, so that it is only valid if the next one is also valid. This can also be applied to several consecutive conditions, chaining them. Conditions may be *or-ed* by setting both (or all) conditions to the same target. Thus, $(a+b)\cdot c \rightarrow target$ is actually deconstructed into its two basic conditions: $a \cdot c \rightarrow target$ and $b \cdot c \rightarrow target$. By the way the conditions are evaluated, when one of these basic conditions is met, the *or-ed* condition is also satisfied in the microprogram.

The maximum number of conditions that can be evaluated in each microinstruction is a parameter specified during the design of the microinstructions. It must be set so that the microinstructions can implement enough conditions to deal with any changes to the original design, while at the same time not overloading the condition evaluation unit in highly demanding applications.

In most cases the number of slots is defined during the design phase to allow for evaluating any condition. In some specific applications, however, this might not be the case, and it may be necessary to consider if evaluating conditions in two or even more cycles is acceptable. In this case the microinstructions will first evaluate a reduced number of conditions, and, if necessary, change the microinstruction sequence to include more microinstructions to complete the evaluation of all conditions. Figure 4.12 shows an example where the evaluation of six conditions with multiple variables

	condition	target	condition	target	condition	target	condition	target	condition	target	actions...
the_state:	if a	eval_a	if b	state_nota_b	if c	state_c	else	eval_notc	...
eval_a:	if b	state_ab	else	state_a_notb	-	-	-	-	-
eval_notc:	if d	state_notc_d	else	state_others	-	-	-	-	-
state_ab:	...				-	-	-	-	...
...									

Figure 4.12: Example of a six-condition evaluation implemented as four-condition microinstructions in two steps. Auxiliary states `eval_a` and `eval_notc` are included to implement a larger number of transitions than those directly supported by the format.

in a single microinstruction is replaced by up to two microinstructions that split the conditions between them. In this case the conditions only have one variable, and not more than four need to be evaluated in each microinstruction. More specifically, two direct transitions may be taken in the first microinstruction, which may also divert to two other microinstructions that evaluate two more conditions each. In most cases splitting conditions like this is possible, unless the timing requirements are very tight.

The second part of the proposed format determines what actions are executed by each microinstruction. The actions consist of driving outputs and updating variables, and are associated to the microinstructions themselves instead of the transitions, making the microprogram behave like a Moore machine. There are two types of actions: outputs and updates of internal variables. Each microinstruction has the number of outputs set in the architecture, and they are managed by selecting the index of the specific output and the desired coded value to be written. Each output has a predefined value set by default at start-up. In Section 4.4.3 the outputs are described in more detail. Internal variables are implemented as counters, explained in Section 4.4.3, which may updated in several ways. In a similar fashion to the outputs, counters are implemented in the microinstruction as an index to each counter and the desired action to apply to it. Each specific value or action is coded to reduce its footprint in the microinstruction format, and the indexes for both outputs and counters are included consecutive in a common list, so it is possible to mix both in the action list.

One of the downsides of this implementation of conditions and actions in the microprogram is that the length of the microinstructions will grow more and more when adding conditions and actions. While it is possible to limit the number of conditions and actions included in each microinstruction, modern FPGAs have enough resources to allow using tens of memory blocks without problems. In Section 4.4.5 the decoding of microinstructions and its resource cost is studied. This opens up the possibility of designing long microprograms with wide microinstructions for most applications without needing to worry about the resource utilization.

4.4.3 Architecture

In this Section we describe a generic architecture in terms of storing and accessing the microprogram, evaluating and updating counters, inputs and

conditions, generating outputs and loading the configuration.

Storage

Microprograms are stored in a number of RAM blocks, usually laid out *horizontally* forming a row. The number of blocks in the rows determine the length of the microinstruction format, so it is possible to increase the number of conditions and actions just by adding more RAM blocks to the microinstruction format. There are two ports in each row of RAM blocks, which allows fetching two microinstructions per cycle. With this implementation it is possible to host two *AND*-super-states concurrently in a single row. If more *AND*-super-states are needed, more rows can be instantiated to implement them.

In this way the microprogram is stored in a number of rows of RAM blocks that are as long as needed by the desired microinstruction length, hosting up to two *AND*-super-states per row. *AND*-super-states include one or more states or super-states, but even if an *AND*-super-state includes a large number of sub-states it should fit in a row, given that RAM blocks are hundreds or even thousands of data words long and are concatenated forming rows.

Access to each *AND*-super-state is granted using one *program counter* (PC) register that addresses the memory directly. In case of implementing history, the PC must be updated in such a way that it keeps record of the history when switching from one *OR*-super-state to another. Hence, each row has a set of history registers that store the entry addresses of OR-states inside an *AND*-super-state. Specific values of the PC are substituted by those stored in the history registers when the jump happens. History is updated when leaving the current OR-state. This is done using the circuit in Figure 4.13, that also shows how history is initialized. Usually *AND*-super-states do not include a large number of OR-states so in the circuit eight history registers are implemented, although it would be possible to implement more for specific applications. Each of these history registers stores the target address for one OR-state. The circuit in Figure 4.14 uses the history registers and two PCs to run concurrently two *AND*-super-states with history in a single row of RAM blocks.

Whenever the microprogram processes a microinstruction a new value

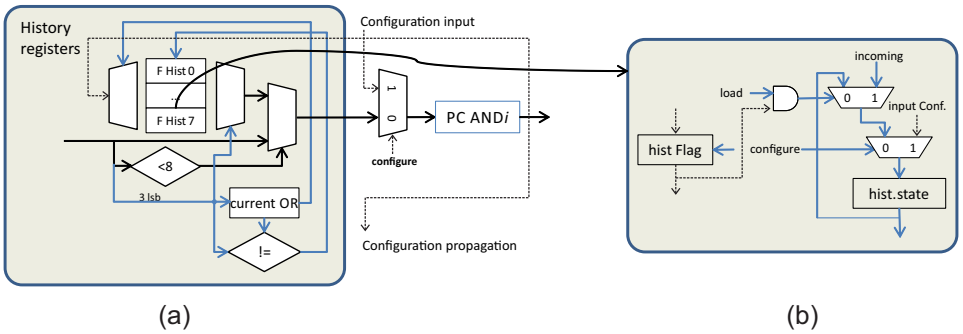

Figure 4.13: History register and its connection to the PC, which is updated when leaving the current OR-state. History is initialized at configuration time as shown in part (b) of the figure.

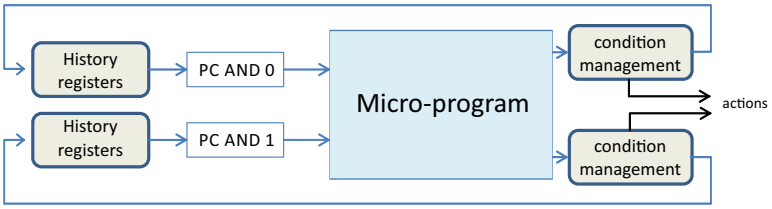

Figure 4.14: Circuit that runs up to two *AND*-super-states. Dual port memory is concurrently addressed by two PCs. Conditions are evaluated and actions are taken independently for both super-states. Transitions are refined using the history and new values for the PC are produced every cycle.

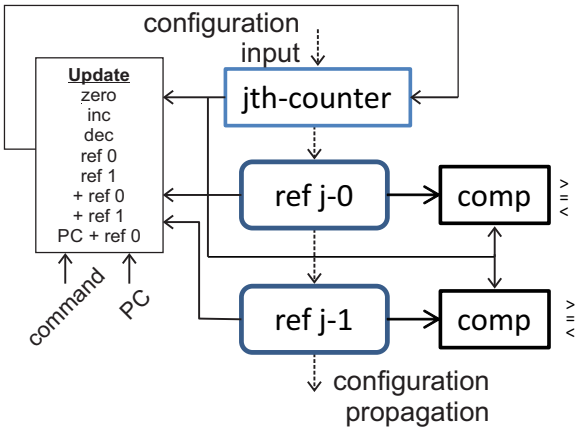

Figure 4.15: Scheme of a counter. At configuration time an initial and reference values are loaded. The counter behaviour is controlled by the active states.

for the PC ($newPC$) that will be used in the next cycle is generated. If the value of $newPC$ is lower than eight, it means that there is a transition to a different OR-state. In this case the history register of the current OR-state is updated, and the target address for the new OR-state is fetched from the history register. The multiplexers and de-multiplexers in Figure 4.13 are used to select the correct register during each operation. Obviously the super-states that are not designed to have history do not update their history register, so the default entry state is always used. This implementation of history is quite soft on its resource utilisation, needing only eight lines in the micromemory.

Counter operation

Counters are implemented as 32-bit registers connected to a 32-bit adder, as shown in Figure 4.15. Counters are initially loaded with their initial value and two reference values. The contents of the counter are updated in a number of ways by selecting the inputs of the multiplexer, which at the same time are driven by the active states. The possible ways are reset, increment, decrement, set to $ref0$ or $ref1$, add $ref0$ or $ref1$, and set to $ref0+PC$, which is useful in special cases as we will show in Section 4.4.4.

The execution of the microprogram will be controlled by the result of

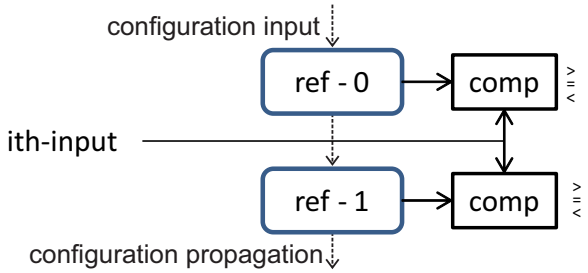

Figure 4.16: Input comparison. At configuration time two reference values are loaded. These values are used to calculate the current value with the reference ones during normal operation.

comparing the value of the counters with their two reference values. These reference values are not only used for the comparison of the counters but also for setting new values to the variables, although usually each counter will only use them in one of those two ways.

Input evaluation

Inputs are implemented in a similar fashion to counters but with a simpler circuit, since their value can not be changed by the statechart itself. This circuit is shown in Figure 4.16, and as counters, it includes two comparators and the needed registers to load and keep the configuration, but without the multiplexer.

Condition chaining

As explained before, the conditions encoded in the microinstructions dictate the transitions between states and the operation of the microprogram. These conditions are included in the microinstruction organized by their priority, so that when they are evaluated in order and one conditions is evaluated true the rest are ignored. Each condition has four fields: the first one is the index of the input or counter that is evaluated by that condition, so its length is set by the total amount of inputs and counters in the microprogram. The second field determines the operation performed by the condition; the options are checking if the value is equal to, different than, lesser, lesser or equal, greater and greater or equal that the first or second reference value. The third field is a single bit (D_i) that indicates if the current condition must

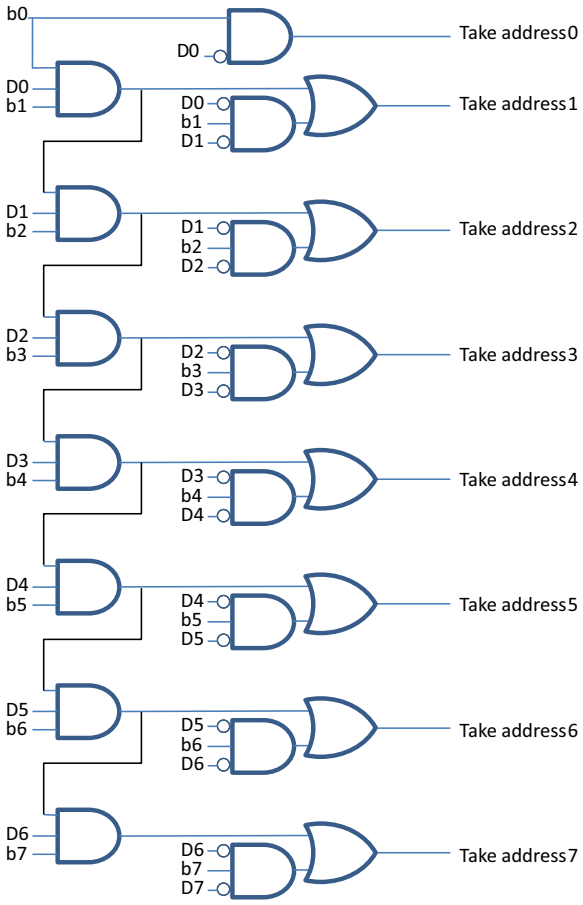

Figure 4.17: Example of chaining eight conditions. Eventually, all the conditions will activate only one output bit, which sets the transition that will take place.

be *and-ed* to the following one (or ones, concatenating conditions with this field set). The last field identifies the address of the next microinstruction if the condition is evaluated true.

Conditions can be chained by the circuit shown in Figure 4.17. In this implementation up to eight conditions can be chained, but the circuit can be fitted to a different number of conditions. In the Figure, the result of each individual evaluation is b_i and the D_i bit indicates if consecutive conditions should be *and-ed*. At the end only one output at maximum will be active, which sets the address of the microinstruction to be used next cycle. This can happen in two ways: if an individual, non-*and-ed* condition is true, *OR* alternatively if all of the previous, *and-ed* conditions are true. This implementation is equivalent to a multiplexer with a decoded selection signal. If all conditions are evaluated false, then the same microinstruction will be repeated next cycle.

Output selection

Outputs work in a similar fashion to counters, with two reference values loaded at configuration time. The microprogram will select the value to be written to an output from a list of the reference values and the content of several counters, controlled by a multiplexer. This is shown in Figure 4.18. The microprograms implement a number of outputs, although not all of them may be used by the system. Even more, among the used ones, only some of them may be activated at the same time, while the other ones should hold their state until their value changes. This is implemented with a register and a multiplexer. The outputs may or not be registered, in a similar fashion to what happens in FPGA's logic blocks. A flag that drives the multiplexer is loaded during configuration, since it is expected that this behaviour does not change during the operations.

Loading configuration

The configuration is loaded word by word using the data network and propagated through the hardware in a serial fashion, in the same way that FPGAs are configured using JTAG. For this to be possible all the memories and registers need to be chained in the propagation of the configuration. This includes the microprogram, the initial content of the history registers,

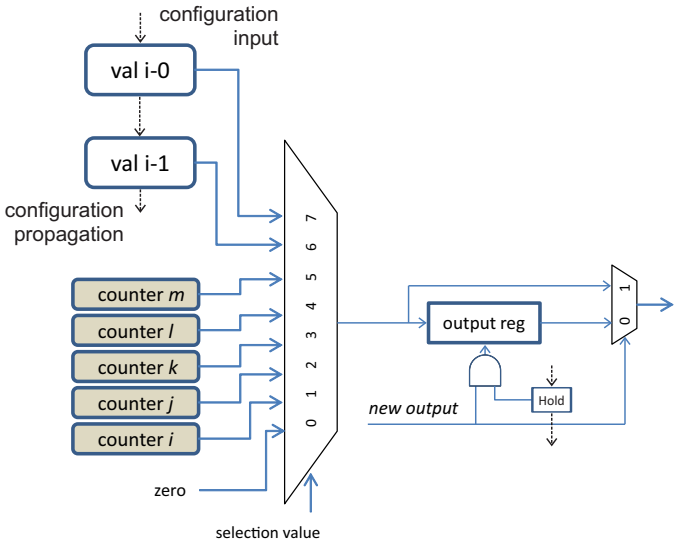

Figure 4.18: Circuit for output selection. The selection value is retrieved from the active microinstructions.

the initial and reference values of the counters and the reference values of the outputs.

Figure 4.19 shows how a microprogram is loaded. There is a configuration counter that is in charge of enabling the write operation to the specified RAM blocks. In the case of the microprogram in the Figure, there are four rows with four blocks each. Each block has a width of 32 bits and a depth of 512 words. Since the input words are 8-bit long, four of them need to be gathered and concatenated before they can be written to the memory blocks. This layout can be changed according to the requirements of each specific microprogram. Figure 4.20 shows a dual port memory block, where each port is used for one *AND*-super-state. The most significant bit is hardwired ('0' for port A, and '1' for port B) to set apart the address space of each *AND*-super-state. Port A is the only used during configuration to write all configuration words.

The configuration of the history registers can be explained using Figure 4.13. An 8-bit register holds the address of each of the eight potential target states, and a flag indicates if the super-state is supposed to implement history or not. The state registers are chained together and the eight flags are implemented together as an extra register chained to the state registers.

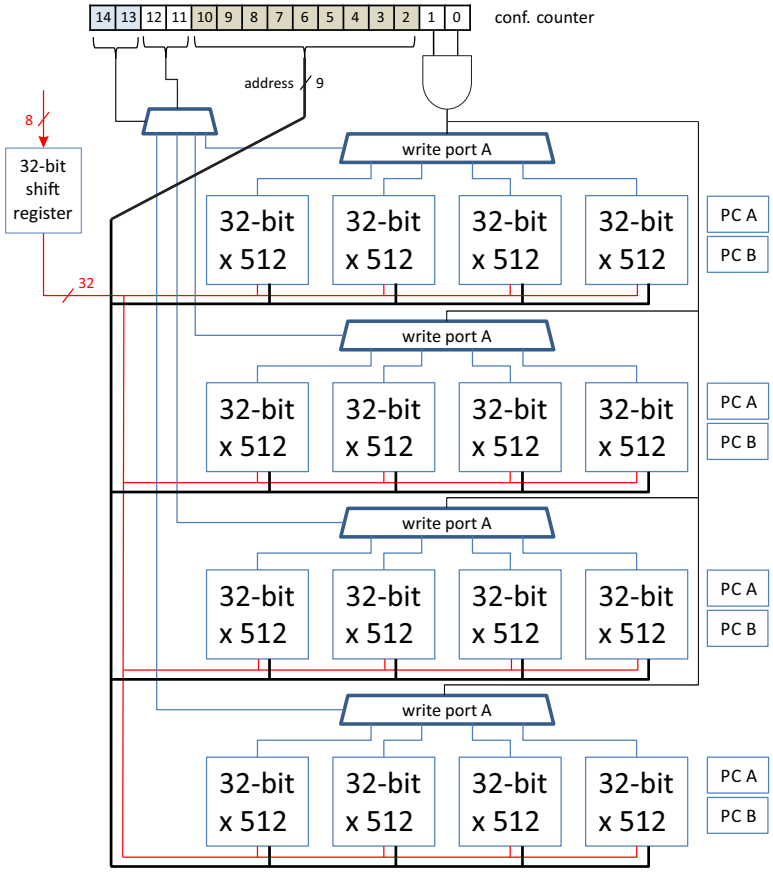

Figure 4.19: Load of a microprogram with four rows of four blocks per row. A simple configuration counter is used to load each word in the right position, row and memory block.

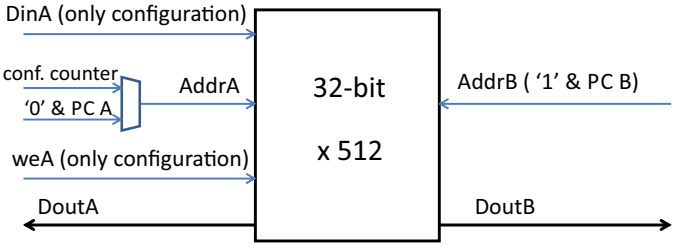

Figure 4.20: Dual port memory block. Two AND-states may be addressed simultaneously using both ports.

The load of the reference values to counters and outputs uses the same chaining method. One 32-bit long initial value and two 32-bit long reference values are loaded for each of the counters and outputs. The micromemory is located at the end of the configuration chain to ensure that when the microprogram is fully loaded, all the other configuration words are loaded as well.

4.4.4 Case example

Let us go back to the statechart in Figure 4.9 to adapt it as an implementation example, and show how it would be adapted as a microprogram using the methodology explained so far. Although the statechart is precisely described the architecture that we will design is open for upgrades.

The statechart that implements the digital watch has several inputs: the four buttons of the digital watch `a`, `b`, `c` and `d`; and five status inputs labelled `chimeOnOff`, `oclock`, `batteryStatus`, `testT1` and `testT2`. The buttons have three different possible values, which are 0 (no action), 1 (pressed) or 2 (released). There are eight counters called `Timer1` and `Timer2` (with configurable reference values, that are set to adjust the behaviour of the watch); `displayUpdate`; `alarmUpdate1` and `alarmUpdate2`; `chimeOnOff`; `countStopWatch`; and `countAlarm`. There are four outputs: `light`, `beep`, `display` (which is actually a complex output that implements all the information shown on the display) and `change`.

Looking at Figure 4.9 seven AND-states have been identified, and rounded up to eight for the implementation and allow for upgrades. This means that four rows of dual-ported memory blocks are needed. The super-states are organised as it is shown in Figure 4.21. The implementation of two of those states, `alarm1` and `update1` in `main/displays/out`, is shown in Figure 4.22. Transitions can take place by pressing the different buttons or by letting the `Timer2` counter roll up for 120 seconds. Regarding the actions, the current microinstruction is tracked with the `alarmUpdate1` counter. Its value is sent to the `display` output to allow the control of the settings of the digital watch. The reference value $ref0$ of the `alarmUpdate1` counter is set to $X - PC(alarm1)$, where $PC(alarm1)$ is the value of the program counter for `alarm1`, so that the `alarmUpdate1` counter sends codes X, $X+1$, $X+2$, etc, to the display. The `change` output is activated ($ref1 = 1$) when some

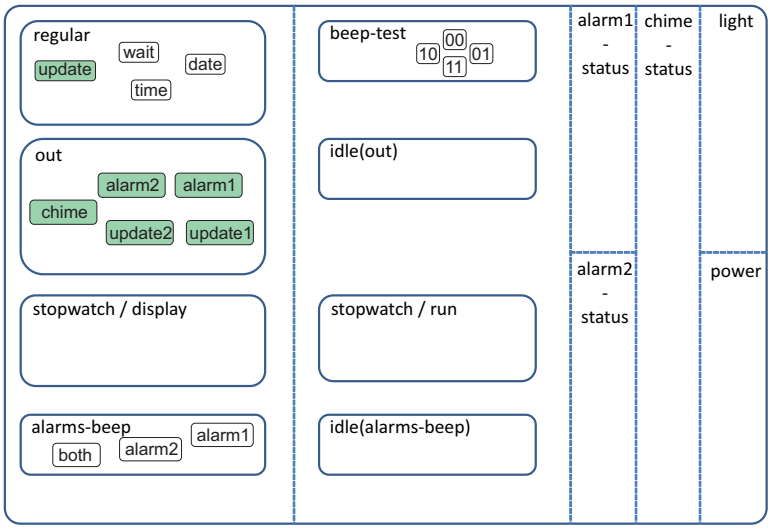

Figure 4.21: Example of microprogrammed organisation of states of the statechart in Figure 4.9. The dashed lines separate the AND-super-states. The leftmost one carries out most of the load, while its neighbour is mainly formed by ghost super-states with the exception of `beep-test` and `stopwatch/run`. The other five AND-super-states are very simple but, hierarchically, are at the same level as the main ones.

mnemonic	condition	target	condition	target	condition	target	condition	target	cnd	tgt	cnd	tgt	action target	action value	action target	action value	aT	aV	aT	aV
alarm1																				
off	Timer2==ref0	time	d==ref0	on	c==ref0	update1	a==ref0	alarm2					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
on	Timer2==ref0	time	d==ref0	off	b==ref0	alarm1	d==ref0	incHr					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
update1	Timer2==ref0	time	c==ref0	10min	b==ref0	alarm1	d==ref0	inc10min					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
Hr	Timer2==ref0	time	c==ref0	Min	b==ref0	alarm1	d==ref0	incMin					**alarmUpdate1**	PC+ref0	**display**	alarmUpdate1	**Timer2**	inc		
10min	Timer2==ref0	time	c==ref0	alarm1	b==ref0	alarm1	d==ref0						**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		
Min	Timer2==ref0	time	Timer2<>ref0	Hr									**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		
incHr	Timer2==ref0	time	Timer2<>ref0	10min									**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		
inc10min	Timer2==ref0	time	Timer2<>ref0	Min									**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		
incMin	Timer2==ref0	time	Timer2<>ref0	Min									**CHANGE**	ref1	**display**	alarmUpdate1	**Timer2**	= 0		

Figure 4.22: Micro-code example for two selected super-states. The format supports up to six transitions and four actions. Some transitions are highlighted with arrows for the sake of clarity. Condition-chaining bits are not shown. Both super-states support history.

value is updated.

In the whole statechart there are nine inputs and eight counters, which are rounded up to sixteen of each and encoded using five bits for the condition evaluations (index). In the digital watch statechart there are no more than four possible transitions from each state, although two more are added to each microinstruction for possible future upgrades. Each transition has five bits for the input/counter index, two bits for the comparisons (*equal*, *greater* and *lesser*), one bit for the reference values (ref_{j0} and ref_{j1}), one bit to invert the comparison, one bit for chaining conditions and eight bits with the target address. This means that the condition evaluation section of the microinstructions has a length of 108 bits (eighteen bits times six conditions).

On top of the conditions there are up to three possible actions for each state of the statechart, which we increased to four to allow upgrades. The actions can apply to the sixteen counters previously implemented and the four outputs, that we increased again to sixteen, so that five bits are also needed to encode the action targets. We proposed eight operations on each counter or output, which can be encoded with three bits. This adds up to eight bits per action, totalling 32 bits for the four actions. Added to 108 bits of the conditions, the total length of the microinstructions is 140 bits. Using 32-bit words on each memory block yields that the memory rows should have a width of five blocks. This means that the actual implementation of the digital watch statechart is quite similar to the one shown in Figure 4.19 but with a more complex counter. The complex counter is needed because of the five memory blocks, which is not a power of two.

To sum up, our proposed implementation of Harel's digital watch using microprograms uses four rows of five memory blocks each; sixteen inputs, sixteen counters and sixteen outputs. Of those, seven inputs, eight counters and twelve outputs are not used and are only intended for future upgrades. One additional AND-super-state could also be implemented in the upgrade, as well as adding several microinstructions to any of the super-states.

4.4.5 Evaluation

The actual amount of resources that a specific microprogram needs to run on an FPGA depends on the number of inputs, counters, outputs and mem-

Component	Logic blocks	Flip-flops	RAM memories
AND-super-states	872	83	20
Counters	247	96	
Inputs	56	64	
Outputs	225	97	
Total	16203	4776	20
Hardwired	131	140	-

Table 4.3: FPGA resources utilisation by our implementation of the digital watch in Figure 4.9.

ory blocks that store the microprogram. Given the numbers obtained in the previous sections, Table 4.3 shows the resources for our implementation, and a comparison to the same statechart implemented in VHDL by hand. One must keep in mind that these numbers are for our presented implementation, and that different implementations will achieve different results. Our microprogrammed implementation also left plenty of room for future upgrades (about half the resources), which are not taken into account on the hardwired implementation. This is noticeable in the large number of resources taken up by big multiplexers that deal with the inputs, counters and outputs. As it can be seen from the comparison to the hardwired implementation, a significant overhead exists for the microprogrammed implementation, but one must take into account that resources are quite inexpensive with current technology, and that the advantages and flexibility of updating the configuration without resynthesizing the design are critical for some applications, such as the miniIOCs at ESS.

Logic blocks or look-up tables (LUTs), which are the pure logic of the hardware, are as expected the most used resource, as a consequence of the large multiplexers needed by the inputs, counters and outputs. The implementation of this design is not a problem in modern hardware, as it can fit in all Xilinx Artix-7 devices except the two smallest ones (12T and 15T), and at least four times in any Xilinx Kintex-7 FPGA [29]. We must also take into account that Harel's statechart is deliberately complex in order to showcase all the functionalities of statecharts. This design had 753 lines of code, compared to the 569 of the handwritten implementation.

We have evaluated the clock speed of this implementation running on the Xilinx xC7A25T and xC7K70T devices using average speed grades. The

	Hand-coded	Microprogrammed
Lines of code	148	689
Logic blocks	24	8416
Flip-flops	43	1872
RAM memories		8
Delay (ns)	1.29	8.56

Table 4.4: Comparison of the resource utilisation for the statechart in Figure 4.1.

minimum speed was 115 MHz, which is enough for most applications. The delay was 8.70 ns, much slower than the handwritten implementation, due to the use of memory blocks and large multiplexers.

The data for the statechart in Figure 4.1 is shown in Table 4.4. It is worth pointing out that, even if this statechart is much simpler than Harel's watch, the number of lines of code is similar in both cases, since many of the components are the same, although different amounts are instantiated in the designs. The number of logic blocks and flip-flops is half compared to Harel's watch while the delay is similar, another example of the overhead that the microprogrammed architecture introduces. In particular the delay is a consequence of the access to the RAM memories and propagating the signals through large multiplexers, whose delay increases logarithmically with the number of inputs.

Chapter 5

Conclusions

In this thesis we have presented the ESS timing system and some contributions that we have made for it. One of them is the standalone mode for the EVRs, very important because of the in-kind, disperse nature of the ESS project. The standalone mode has permitted the test of the ESS systems and devices all over Europe without deploying a complete timing system in each location, which would have multiplied the price of the deployments. It has also allowed to test different subsystems concurrently and independently of each other even in the same location, since it has been possible to keep one subsystem running while another is down, without depending on a single node that affects all the different subsystems.
A proof of concept of the miniIOC was developed by the same author of this thesis but before it started. Later during this thesis the description and specification of the miniIOC was developed and refined from the proof of concept, and its requirements gathered. An in-kind partner is developing the miniIOC according to the specifications provided. The miniIOC is expected to be a low-cost solution for providing synchronization in constrained or difficult to access spaces.
The supercycle application, whose specification and requirements where defined during this thesis, will be a crucial tool for commissioning of the accelerator and target. It will allow for ramping up the proton beam steadily in an automated way.

The events and data items are the fundamental pieces of the ESS timing system. During this thesis the timing requirements for each of the ESS systems were gathered and an implementation strategy was developed. Later the event and data items lists were created fulfilling the requirements and

strategy. Also the EPICS databases that organize the data items in the data buffer for easy access and availability were written, and some basic event sequences with their respective delays were generated. These lists are already being used by some subsystems.

A tool for automatically implementing statecharts as VHDL code was written. It will allow for reducing the manpower needed to deploy and update miniIOCs with timing capabilities when they are available, although it can be used for any other system that can be described as a statechart. The tool has proven to be able to generate synthesizable code that is free of errors, although somewhat more verbose than the equivalent hand-written VHDL code. Regarding the resource utilisation, however, there is little difference between the code generated by the tool and the hand-written one. This tool will be used for creating the firmware for the miniIOC and embedded EVRs.

An alternative methodology to the previous tool has also been developed, based on microprogramming. This methodology would allow upgrading the functionality of FPGA-based devices or even ASICs without re-synthesising any code, just the upgraded microcode would need to be loaded in the device. This would allow keeping the system up-to-date even if the target devices or design tools are not supported any more. The amount of resources of this implementation compared to the hand-written would be, however, one to two orders of magnitude higher. This should not be an issue with the amount of resources present in modern devices.

5.1 Future work

There is still room for some improvements in the ESS timing system. One of them is the EPICS layer support for delayed gated signals in the EVRs and for backplane lines inputs in the mTCA-EVR-300(U), which are available in the latest version of the firmware.

Regarding the tool for generating HDL code from a statechart description, a number of improvements are planned:

- New versions of Yakindu can include priority in the transitions, so if several events (or conditions) that trigger different transitions from a unique state are raised, the statechart can determine exactly which transition should take place. In previous versions of Yakindu it was

the designer's responsibility to ensure that no such situations would happen, or the statechart would behave in an undefined way. This priority specification gets reflected in the XML file that describes the statechart and that is used as the input for our tool. When we started designing our tool priority was still not supported in Yakindu, so our tool does not support it either. In future versions of our tool priority can be implemented.

- Support for different HDL languages, such as Verilog (at the moment our tool can only create VHDL code). This would be done by extracting the lines that write to the output file to a centralised place and replacing them with macros or string-type variables that can then be easily configured in only one place.

- Remove the requirement that only allows statecharts without transitions between different regions. This new version for the tool has already been planned. The upgrade would require that each VHDL process monitors all of the transitions in the statechart, and not only those transitions in the regions of the hierarchy tree branch where the current region is.

- Support for deep history. In the current version our tool only implements shallow history, which means that a history node needs to be added to each region in all levels of the hierarchy that require to resume the latest running state when the region is re-activated. With deep history support only one history node would be needed, that would remember the running state in all levels of the hierarchy within the region where the history node is. Although the lack of deep history does not limit in any way the functionality of statecharts, since an equivalent statechart can always be created using only shallow history, deep history allows creating simpler statecharts.

Regarding the microprogrammed based methodology, we intend to develop a protocol with which it would be possible to update the microcode using the data connection. This methodology could also be included in our tool, so that the tool could create the firmware for the microprogrammed deployment.

5.2 Publications derived from this thesis

Here we present a list of publications derived from this thesis:

- J. Cereijo García, T. Korhonen, J. H. Lee, R. R. Osorio, and D. Piso Fernández, "Timing system at ESS", in *8th Int. Particle Accelerator Conf. (IPAC17)*, 2017.
- J. Jamroz, J. Cereijo García, T. Korhonen, and J. H. Lee, "Timing system integration with MTCA at ESS", in *17th Biennial International Conference on Accelerator and Large Experimental Physics Control Systems, 2019*.
- J. Cereijo García and R. R. Osorio, "Hardware implementation of statecharts for FPGA-based control in scientific facilities", in *XXXIV Conference on Design of Circuits and Integrated Systems, DCIS*, pp. 1-6, 2019.
- J. Cereijo García and R. R. Osorio, "A microprogrammed approach for implementing statecharts", in *Euromicro Conference on Digital System Design, 2019*, pp. 27-34, 2019.
- J. Cereijo García and R. R. Osorio, "Comparison of hardwired and microprogrammed statechart implementations", in *MDPI Electronics*, 2020.

Appendix A

Beam modes

Table A.1 shows a list of the ESS beam modes and their characteristics.

Table A.1: ESS beam modes. Source: [2].

#	Name	Beam envelope	Description	Average power at 2 GeV
0	None	No beam	No beam	
1	Probe beam	0 to 5 μs, 0 to 1 Hz, 6 mA	First beam through a particular section; non-damaging even in the case of total beam loss (even repeated); used to verify that machine configuration is not grossly incorrect	60 W
2	Fast Tuning	0 to 5 μs, 0 to 14 Hz, 0 to 62.5 mA	Limited beam loading; used for fast scans to rapidly determine/verify RF setpoints and measure beam profiles with wire scanners	8.5 KW
3	Slow Tuning	0 to 50 μs, 0 to 1 Hz, 0 to 62.5 mA	Longest pulses that allow operation of invasive proton beam instrumentation devices like wire scanners; long enough beam pulses to diagnose and monitor RF feedback and the onset of beam loading; used to perform more precise single-pulse measurements	6 KW
4	Fat Probe	0 to 5 μs, 0 to 1 Hz, 0 to 62.5 mA	Very short pulse to be used during Warm Linac commissioning. It would allow installation of the rest of the Linac in parallel to Beam Commissioning. It would not be used after commissioning	600 W
5	Slow Studies	0 to 50 μs, 0 to 0.1 Hz, 0 to 62.5 mA	Short pulse to be used during Warm Linac commissioning. It would allow installation of the rest of the Linac in parallel to Beam Commissioning	600 W
6	Long Pulse Verification	0 to 2.86 ms, 0 to 1/30 Hz, 0 to 62.5 mA	Only used when machine reasonably tuned to the tuning dump or the target; slowly-increasing pulse lengths are used to tune RF feedforward, verify beam loading and Lorentz force detuning compensation, and tune for low beam losses. If possible, intermediate short pulses at 1Hz should be supplied to monitor stability between long pulses	15 KW
7	Shielding Verification	Low power ($\sim$30 kW), nominal energy and peak current	To be defined better once the exact requirements for the shielding verification (power, pulse length) are known	$\sim$30 kW
8	Production	2.86 ms, [1-14] Hz	Production	5 MW

Appendix B

Beam destinations

Table B.1 shows a list of the possible destinations of the proton beam at ESS.

Table B.1: ESS beam destinations.

Number	Beam destination
0	No destination (initial state)
1	LEBT Faraday Cup (FC)
2	MEBT FC
3	DTL 2 FC
4	DTL 4 FC
5	Spokes FC
6	MBL Beam Stop
7	Tuning Dump
8	Target

Appendix C

Resumen en castellano de esta tesis

Las instalaciones industriales y científicas se están volviendo cada vez más complejas para resolver nuevos desafíos y volverse más eficientes. Esto afecta al equipamiento que forma parte de estos centros, pero también, y quizás aún más importante, los sistemas que integran todos estos dispositivos para que puedan trabajar conjuntamente para cumplir con el objetivo de la instalación. Uno de estos sistemas es el de control, que es responsable de administrar el resto de sistemas para que funcionen de la manera correcta y al unísono. Los sistemas de control son redes complejas formadas por hardware, software y sus respectivas configuraciones, y que se integran entre sí para controlar con éxito la máquina o instalación. Estos sistemas de control aumentan su complejidad según también se incrementa la de los dispositivos que controlan, y lo hace de forma exponencial cuando el número de equipos y sistemas aumenta, ya que todos ellos deben de ser administrados y configurados de forma compatible y conjunta. También es muy importante que los sistemas de control sean implementados sin errores, ya que la alta complejidad de estos sistemas hace que la probabilidad de errores aumente, que sean más difíciles de solucionar y que tengan consecuencias más graves, que pueden incluso evitar el funcionamiento de la instalación, dañarla o incluso herir a personas. Por esta razón está aumentando el uso de herramientas que automatizan y simplifican el diseño, desarrollo, instalación y puesta a punto de los sistemas de control, para conseguir centros mejores, más seguros y más fiables.

Un tipo de centros que necesitan avanzados sistemas de control son las

fuentes de neutrones. Estas instalaciones producen neutrones que son muy útiles para la investigación en diversas disciplinas como la ingeniería de materiales, física fundamental, productos farmacéuticos, etc. Esto es debido a que los neutrones tienen ciertas características especiales como que su energía y escala de longitud se pueden ajustar para procesos atómicos y moleculares, que se pueden usar como sondas débilmente acopladas, y que pueden penetrar de forma profunda y sin causar daños en las muestras bajo estudio, que dispersa los neutrones en haces que crean patrones específicos.

La European Spallation Source (Fuente Europea de Espalación) es una instalación científica para la investigación con neutrones que está actualmente en fase de diseño y construcción en Lund, en el sur de Suecia. Una parte de la European Spallation Source, el Data Management and Software Centre (Centro de Gestión de Datos y Software) está situada en Copenhague, en Dinamarca, donde los datos producidos por los experimentos se enviarán y serán procesados y almacenados. Cuando esté terminada y entre en funcionamiento, la European Spallation Source será la fuente de neutrones más grande del mundo. Está formada por un acelerador lineal que dispara un haz de protones a un blanco que produce neutrones por el proceso de espalación. Los neutrones son luego guiados a un conjunto de experimentos que se utilizan para realizar investigación en diferentes disciplinas. La European Spallation Source es una colaboración entre varios países europeos, que diseñan y construyen las diferentes partes y sistemas por toda Europa como un proyecto en especie, y que envían las distintas partes y sistemas a Lund, donde se ensamblan juntas.

El entorno software que se usa en la European Spallation Source es EPICS, el Experimental Physics and Industrial Control System (Sistema de Control Industrial y de Física Experimental). EPICS es de código abierto y desarrollado por la misma comunidad de instalaciones científicas e industriales. En cuanto al hardware utilizado en la European Spallation Source, la mayor parte está en el factor de forma μTCA, especialmente aquellas aplicaciones que requieren un procesado rápido, de hasta cientos de MBytes por segundo. Uno de estos sistemas es el sistema de temporización. El sistema de temporización se encarga de sincronizar todos los dispositivos entre sí y con el mundo exterior a la instalación o centro. Esta sincronización con el mundo exterior generalmente se realiza conectándose a una fuente externa, como el GPS (Global Positioning System o Sistema de Posicionamiento Global), del cual el sistema de temporización deriva su referencia de tiempo y que utiliza para sincronizar la instalación. Los sistemas de temporización generalmente

tienen un nodo maestro que define la referencia temporal absoluta para el centro, disciplinada por la fuente externa, y la envía al resto del sistema de temporización, formado por nodos esclavos. El nodo maestro también es el punto central desde donde se gestionan la operación y la configuración del sistema de temporización, y que codifica esta configuración para enviarla a los nodos esclavos y que éstos la puedan utilizar. Los nodos esclavos reciben la referencia temporal y la información enviados por el nodo maestro y reaccionan según estén configurados para ello. Las reacciones típicas son el envío de señales de activación precisas y registro temporal de diferentes señales y acontecimientos.

Hay una serie de desafíos que el sistema de temporización necesita superar para cumplir sus objetivos correctamente. La más importante es la sincronización de la instalación. Para lograr la sincronización, es necesario definir un reloj maestro común y enviarlo al resto de los nodos del sistema de temporización. Si esto no se hace, y se decide que cada uno de los nodos mantenga su propio reloj, cada uno de ellos con la misma frecuencia, aparece un problema enseguida: incluso si los relojes están desfasados en sólo una fracción de una parte por millón, dada la frecuencia con la que funciona el sistema de temporización, generalmente alrededor de 100 MHz, en cuestión de unos pocos segundos los nodos se desfasarán entre sí en más de un ciclo de reloj. La única solución para evitar esto es definir y distribuir un único reloj maestro común que sea compartido por todos los nodos. Además de definir el reloj común, es también necesario definir un momento en el tiempo que actúe como referencia temporal y distribuir ese momento sin retardo a todos los nodos del sistema. Como esto no es posible, especialmente dado que los nodos que necesitan sincronización están muy lejos unos de otros, en algunos casos a cientos de metros o incluso a kilómetros de distancia, se necesita una solución alternativa, que implica calcular el retardo de la transmisión entre los nodos para poder calcular luego la referencia temporal original en todos los nodos. De esta forma forma se consigue que los nodos compartan la misma frecuencia, ya el reloj que todos ellos usan es el mismo, y la misma fase, después de calcular y corregir el retardo, todo ello dentro de unos rangos de desviación aceptables.

Para compartir el reloj entre todos los nodos del sistema de temporización, en concreto el reloj que define el nodo maestro, se usan lazos de seguimiento de fase, o PLLs por sus siglas en inglés (Phase-Locked Loops). Estos dispositivos reciben como entrada la señal del nodo maestro y mediante un lazo de realimentación varían su salida hasta que ésta coincide con

la entrada. En este momento el lazo de seguimiento de fase está sincronizado con la entrada, y su señal se utiliza internamente en los nodos esclavos del sistema de sincronización. Esto también permite la recuperación del reloj de entrada.

Existen una serie de tecnologías disponibles actualmente para la sincronización de sistemas. Las más importantes son el Network Time Protocol (NTP) que consigue sincronización hasta el nivel de milisegundos, el Precision Time Protocol (PTP) hasta microsegundos y, ya en el rango de nanosegundos, White Rabbit y el sistema de temporización de Micro-Research Finland (MRF). En concreto el hardware del sistema de Micro-Research Finland es el que se emplea en la European Spallation Source.

El sistema de temporización de la European Spallation Source está formado por un nodo maestro, llamado generador de eventos, que envía eventos y otra información de sincronización y temporización a los receptores de eventos, que reaccionan a estos eventos de la forma en la que están configurados para ello, principalmente enviando señales de activación perfectamente sincronizadas a los diferentes dispositivos que forman parte de la European Spallation Source. En el contexto del sistema de temporización, un evento es una señal momentánea y enumerada. Además de distribuir eventos y enviar las señales de activación a los diferentes dispositivos, el sistema de temporización también distribuye una serie de relojes obtenidos a partir del reloj maestro común mediante división entre números enteros positivos y un *buffer* con información relacionada con el haz de protones y que se actualiza en cada ciclo del haz. También proporciona toda la funcionalidad necesaria para asignar etiquetas de registro temporal a todos los eventos y sucesos importantes que ocurren en la European Spallation Source, de forma que todas las etiquetas están referidas a un sistema de referencia temporal común, para poder después comparar toda la información y sucesos entre ellos. Micro-Research Finland proporciona la mayoría de los nodos del sistema de temporización de la European Spallation Source. Este hardware se ha empleado en múltiples otras instalaciones con muy buen rendimiento.

El reloj maestro común que utiliza el sistema de temporización, de 88.0525 MHz, se obtiene a partir de dividir la señal de radio frecuencia que alimenta el equipamiento de aceleración del haz de protones entre cuatro, de forma que las señales enviadas por el sistema de temporización están sincronizadas con la señal de radio frecuencia. Además en cada ciclo de este reloj maestro se envía un único evento de ocho bits (por lo que hay 256

diferentes eventos posibles) y bien un *byte* de ocho bits del *buffer* con la información relacionada con el haz de protones y que se copia del generador de eventos a todos los receptores de eventos o bien una muestra de cada uno de los ochos relojes derivados del reloj maestro. El *buffer* de datos y los relojes derivados se envían en ciclos alternativos, de forma que en un ciclo se envía un evento y un *byte* del *buffer* y en siguiente ciclo otro evento y una muestra de los relojes, y así consecutivamente. A esta transmisión se le aplica además el protocolo 8b/10b, de forma que la frecuencia real de la transmisión entre el generador de eventos y los receptores de eventos del sistema de temporización es veinte veces la frecuencia del reloj maestro. Además el sistema de temporización se encarga de crear la estructura del haz de protones, que tiene una longitud máxima de 2.86 milisegundos y una frecuencia de repetición de 14 Hz. Todas estas frecuencias y tiempos los crea el sistema de temporización a partir del reloj maestro común.

Los eventos que envía el sistema de temporización pueden tener varios orígenes: una lista con una secuencia de eventos y el retardo entre todos ellos presente en todos los nodos del sistema (aunque por lo general sólo la secuencia del generador de eventos se envía a toda la red, las secuencias de los receptores de eventos son normalmente sólo locales), unos contadores multiplexados presentes sólo en el generador de eventos, canales de entradas presentes en todos los nodos, y por último también es posible forzar la activación de un evento por software. Además se envían también de forma automática los eventos que emplea el mecanismo de etiquetado temporal de datos. Estas etiquetas tienen dos partes: un número que es el tiempo UNIX, enviado una vez por segundo desde el generador de eventos a los receptores de eventos estando sincronizado a un receptor GPS, y una segunda parte local a cada receptor de eventos y que es un simple contador que cuenta ciclos del reloj maestro. De esta forma la granularidad de las etiquetas de referencia temporal es de aproximadamente 11.357 nanosegundos.

El hardware del sistema de temporización producido por Micro-Research Finland empleado en la European Spallation Source son el mTCA-EVM-300, que combina en una misma tarjeta un generador de eventos y un módulo de distribución múltiple y el mTCA-EVR-300(U) y el PCIe-EVR-300DC, estos dos últimos receptores de eventos. Los receptores de eventos cuentan con una memoria de acceso aleatorio (RAM por las siglas en inglés de Random Access Memory) que indica cómo reaccionar a cada evento, generadores de pulsos que crean señales de activación configurables a partir de los eventos, divisores que crean relojes derivados del reloj maestro, diferentes canales

de entrada y salida, el mecanismo de etiquetado de referencias temporales, un *buffer* de eventos y el *buffer* de parámetros relacionados con el haz de protones.

La integración del sistema de temporización en el sistema de control basado en EPICS está hecha con el módulo de EPICS *mrfioc2*. Este módulo cuenta con los módulos del kernel, mapas de registros, soporte de dispositivos y las tablas de datos necesarios para controlar todos los nodos del sistema de temporización.

El sistema de temporización tiene una serie de requisitos que vienen dados por los sistemas que necesitan temporización y sincronización en la European Spallation Source. Básicamente estos requisitos tienen que ver con la sincronización de los distintos dispositivos y las interfaces del sistema de temporización. Los principales sistemas que utilizan la información distribuida por el sistema de temporización son los detectores de neutrones, los *choppers* que dan forma a los haces de neutrones, el blanco rotatorio que libera neutrones, los sistemas de aceleración de protones, la fuente de protones, los *choppers* de protones, los sistemas de protección y la instrumentación del haz de protones.

Además del estudio, implementación e integración del sistema de temporización para la European Spallation Source, en esta tesis se han hecho una serie de contribuciones para este sistema de temporización. Entre las más importantes destaca un nuevo modo de funcionamiento de los receptores de eventos, de forma que éstos puedan trabajar independientemente de un generador de eventos, pero aun así generando eventos y etiquetando temporalmente los datos. Éste ha sido un modo de funcionamiento muy usado en el diseño de la European Spallation Source debido a su naturaleza como proyecto en especie, ya que ha permitido instalar sistemas de temporización locales sólo formados por un receptor de eventos, lo que ha abaratado los costes de diseño de forma distribuida de la European Spallation Source por toda Europa. En concreto durante el desarrollo de esta tesis en esta instalación se ha detectado la necesidad de este modo, del que se han escrito los requisitos que se han enviado a Micro-Research Finland, que los ha implementado en el firmware de los receptores de eventos. De vuelta en la European Spallation Source este método se ha integrado en *mrfioc2*, se ha testeado y se ha escrito documentación y ejemplos de uso.

Otra contribución hecha durante esta tesis para el sistema de tempo-

rización es el desarrollo, entre la European Spallation Source y un centro colaborador, de una tarjeta con un receptor de eventos empotrado que se empleará para ofrecer los servicios del sistema de temporización en localizaciones remotas o de difícil acceso, de forma que no sea necesario instalar un receptor de eventos de Micro-Research Finland y todo su equipamiento acompañante en estas localizaciones. Esta tarjeta está basada en una Zynq de Xilinx, que incluye en un mismo sustrato una FPGA y un procesador. La FPGA es necesaria para recibir y procesar en tiempo real, y aun más importante, de forma determinista, las señales del sistema de sincronización y actuar en consecuencia, creando señales y sincronización y etiquetando temporalmente distintos datos y señales. El procesador es necesario para integrar esta tarjeta en el sistema de control de la European Spallation Source. Además de esto la tarjeta cuenta con un conector FMC (FPGA Mezzanine Card) para ofrecer flexibilidad en las interfaces, pudiendo instalar en este conector módulos con la interfaz deseada.

También se ha ideado una aplicación de alto nivel que se utilizará durante los procesos de puesta a punto de la instalación de la European Spallation Source y durante los procesos de encendido, que permite definir una serie de secuencias de eventos y de parámetros relacionados con el haz de protones que se ejecutan automáticamente y de forma sincronizada una después de otra, permitiendo controlar al detalle cómo funciona la instalación para ir progresando poco a poco comprobando constantemente que todos está sucediendo como se espera. El diseño en sí no forma parte de esta tesis, pero sí forma parte la recogida de los requisitos para esta aplicación de alto nivel, el diseño de la interfaz entre la herramienta y el sistema de temporización y la colaboración para el diseño del motor que traduce las características de los haces de protones deseados por los operadores a eventos y parámetros incluidos y distribuidos por el sistema de temporización.

También durante el desarrollo de esta tesis se han estudiado los requisitos de todos los sistemas de la European Spallation Source y se han diseñado la estrategia y las listas de eventos y de parámetros relacionados con el haz de protones, de forma que estas listas sean compatibles con el sistema de temporización. Se han implementado estas listas en el protocolo que utiliza el sistema de temporización, y se ha desarrollado una estrategia para implementar el control y la configuración del generador de eventos y los receptores de eventos, de forma que todos éstos entiendan y reaccionen correctamente a estas listas.

Lo que no se incluye en el diseño de la tarjeta con el receptor de eventos empotrado antes mencionada es el diseño del firmware que implemente la funcionalidad del receptor de eventos. Teniendo en cuanta las características de la propia tarjeta y el uso que se va a hacer de ella, hay una serie de cosas que hay que tener en cuenta a la hora de diseñar este firmware. En primer lugar el propio conector FMC hace que se necesite mucha flexibilidad en el firmware, o al menos que cambiarlo y actualizarlo sea muy sencillo. Otra es que el firmware debe de usar la menor cantidad de recursos posible. Mientras que los receptores de eventos normales trabajan conjuntamente con otros dispositivos, de forma que cada dispositivo está especializado en una sola tarea, en el caso de la tarjeta con el receptor de eventos empotrado este dispositivo debe de realizar por sí mismo un gran número de tareas muy diferentes entre sí. Alguna de estas tareas puede que necesite recursos de la FPGA, por lo que es necesario que la ocupación de los recursos de la FPGA por parte del sistema de temporización debe ser lo menor posible, llegando incluso a tener implementaciones específicas para cada función de la tarjeta. Todo esto hace que sea necesario implementar e instalar diferentes firmwares de forma rápida y sencilla, y con el menor número posible de errores. Esta última parte es muy importante, ya que al reimplementando y actualizando el firmware constantemente, la probabilidad de errores crece. Por esta razón, y como parte de esta tesis, se han desarrollado una herramienta y una metodología para implementar sistemas de control, tal y como el sistema de temporización, en FPGAs de forma rápida y reduciendo al máximo el número de errores.

Tanto la herramienta como la metodología están basadas en statecharts, una ampliación de máquinas de estados finitos que permiten describir sistemas de forma gráfica en base a estados y transiciones, desarrollada por Harel. Al contrario que las máquinas de estados finitos, donde sólo un estado puede estar activo en cualquier momento, las statecharts pueden tener varios estados activos a la vez. Esto es posible gracias a una serie de características de las statecharts, entre las que destacan la ortogonalidad o concurrencia, la profundidad o jerarquía y la historia. Estas características son las que solucionan el principal inconveniente de las máquinas de estados finitos, que es que su complejidad crece enormemente cuando se añaden más y más estados y transiciones. La ortogonalidad o concurrencia de las statecharts permite definir varios estados que están activos a la vez de forma paralela, y que funcionan de forma casi independiente. La profundidad o jerarquía permite agrupar estados o incluso máquinas de estados completas dentro de otro estado, de forma que actuando sobre el estado padre (ac-

tivándolo o desactivándolo, es decir, realizando transiciones que lleven a él o salgan de él) se actúa a la vez sobre toda la parte de la statechart incluida en él, todo esto de forma sencilla. Los estados que incluyen otros estados dentro de ellos se llaman super-estados. Por último la historia permite un mejor control del funcionamiento del sistema, permitiendo que se retomen acciones o situaciones tal y como estaban ocurriendo antes de que se desactivaran. También, y gracias a que las statecharts son representaciones gráficas, son muy sencillas de comprender.

En esta tesis también se analiza el estado del arte en cuanto a la implementación en hardware de statecharts de forma automática. Aunque hay numerosos estudios y aplicaciones que han intentado realizar esta implementación automática, todos ellos sufren de diferentes problemas, principalmente en lo que se refiere a la historia, que nunca se llega a implementar satisfactoriamente, o a que la implementación es parcial y aun se necesita un alto dominio de lenguajes de descripción hardware para completar la implementación. Comparada con estos estudios previos, nuestra herramienta es capaz de crear un archivo VHDL listo para ser sintetizado e implementado a partir simplemente de la representación del sistema con formato de statechart, e incluyendo todas las características de las statecharts, incluso la historia. El objetivo de esta herramienta es crear el firmware para la tarjeta con el receptor de eventos empotrado, siendo muy fácil su actualización ya que sólo incluye modificar la statechart del sistema. Después de eso la herramienta crea el fichero VHDL actualizado. Aunque nuestra herramienta nació con esta tarjeta como objetivo principal, es capaz de crear código VHDL para cualquier sistema que sea representado como una statechart.

La herramienta usa como entrada un fichero XML que se puede crear con Yakindu, una herramienta para diseñar statecharts gráficamente. La herramienta está basada en el parseador Xerces-C++ para interpretar el código XML y producir el código VHDL. Nuestra herramienta condiciona un poco la statechart a implementar, principalmente por el hecho que no se permiten transiciones entre estados en distintos niveles de la jerarquía. Esta restricción no es demasiado estricta ya que la statechart se puede modificar para que tenga la misma funcionalidad con una representación un poco distinta, y además desaparecerá en futuras versiones de la herramienta que ya están en desarrollo. También se limita a las características de las statecharts originales definidas por Harel, ignorando nuevas características incluidas en Yakindu.

La estrategia que emplea la herramienta se basa en definir un proceso de VHDL por cada región, tal y como las define Yakindu, de la statechart. De esta forma se implementa la ortogonalidad, ya que los procesos de VHDL son concurrentes. La profundidad o jerarquía se implementa con todos los procesos de VHDL incluyendo en su lista de activación todos los eventos que pueden provocar transiciones, y para cada región reaccionando no sólo a los eventos dentro de esa región, si no a todos los eventos que actúan sobre los niveles superiores de la jerarquía. De esta forma las regiones reaccionan correctamente. La historia se implementa usando un pseudo-estado al que transicionan las regiones cuando se sale de esa región o se desactiva. Si una región tiene historia, además se guarda en un registro el valor de un enumerador que define en qué estado se estaba ejecutando la región cuando se salió de ella. Este pseudo-estado es el nodo de entrada de cada región, y que por defecto transita a un estado concreto, que es el estado de entrada; en el caso de que la región implemente historia, el estado dictado por el enumerador sustituye al estado por defecto. Esto significa que los procesos VHDL siempre están activos en todo momento, aunque cuando la región está desactivada lo hagan en el pseudo-estado. Además de implementa una generación distribuida de salidas y acciones y condiciones tanto relacionadas a las transiciones como a los estados.

La herramienta necesita parsear el fichero XML que describe la statechart varias veces para crear todas las partes del fichero VHDL de salida, de forma que éste sea correcto y sintetizable. Esto comienza con la inicialización del parseador, la declaración de la entidad VHDL y la definición de los tipos VHDL y las señales. Se crea un tipo para cada región, cuyos valores posibles son los estados en esa región, y dos señales, que representarán el estado actual y próximo en esa región. También una tercera señal para el registro de historia si es necesario. Después se realiza creación de los procesos VHDL como se ha explicado antes, que es el paso más complejo que realiza la herramienta. El estado actual en cada región se define con estructuras *case... when...*, mientras que las transiciones en cada estado se definen con estructuras *if... then... else...*. También las acciones y la historia se definen en este paso. Finalmente se define un proceso de sincronización que actualiza el estado actual con el valor del próximo estado, tal y como lo define la señal asociada.

Los archivos VHDL creados por nuestra herramienta son más prolijos que el código VHDL equivalente para la misma statechart escrito a mano, teniendo alrededor del doble de líneas. Sin embargo en cuanto a los recursos

usados por ambos códigos cuando se implementan sobre una FPGA, hay muy poca diferencia entre ambos.

Aunque actualmente la herramienta sólo produce código VHDL, es posible modificarla que produzca código en otros lenguajes de descripción hardware.

La metodología que hemos ideado, en lugar de producir nuevo código que necesita ser sintetizado e implementado tras cada actualización o cambio de la statechart, se basa en crear una implementación microprogramada de la statechart con espacio para ampliaciones, de forma que con esta metodología sólo se necesita escribir el código VHDL, a mano, una vez, y después las actualizaciones se realizarían simplemente cargando las nuevas microinstrucciones sobre el firmware que ya estaba instalado.

La estrategia que sigue esta metodología se basa en implementar los super-estados concurrentes como microprogramas independientes, de forma que se implementa un número de microprogramas que coincide con el máximo número de estados que pueden estar activos a la vez. Cuando la statechart está en una situación con un número de estados activos menor que este máximo, ciertos microprogramas están ejecutando microinstrucciones fantasma. Estas microinstrucciones fantasma siguen las transiciones normales, siguiendo o copiando a otras microinstrucciones que sí están activas, pero sin tomar acciones o escribir en contadores o salidas. Cuando la statechart, en su comportamiento normal, transiciona a una situación en la que más estados deben estar activos, las microinstrucciones fantasma transicionan a microinstrucciones normales, en las que sí se ejecutan acciones. En cuanto a las transiciones simples como las que se dan en las máquinas de estados finitos, en la arquitectura microprogramada se corresponden con cargas de nuevas microinstrucciones que implementan los nuevos estados. En cuanto a la historia, sólo se mantiene al nivel más bajo.

El formato de las microinstrucciones está formado por dos partes: la primera es la evaluación de condiciones y la segunda es la ejecución de acciones. En cuanto a las condiciones, se evalúan una a una hasta que haya una que sea cierta, y el resto se ignoran. Se pueden concatenar condiciones con un bit de concatenación. El número máximo de condiciones se fija al diseñar el formato de las microinstrucciones, aunque se pueden dividir condiciones entre dos o más microinstrucciones. Las acciones se evalúan sobre entradas y contadores, expresadas de forma indexada en el formato de las microin-

strucciones, que también cuentan con un campo con la microinstrucción de destino. La segunda parte, la ejecución de acciones, cuenta igual con un campo para el índice de contadores y salidas y otro para la acción a ejecutar sobre ellos. Igualmente el número máximo se fija al diseñar la microinstrucción.

Las microinstrucciones se guardan en las memorias de acceso aleatorio de las FPGAs, organizadas horizontalmente formando filas tan largas como sea necesario por el formato de las microinstrucciones. Gracias a que los bloques de memoria actuales suelen contar con dos puertos es posible cargar dos microinstrucciones de cada bloque. Si la statechart puede llegar a tener más de dos super-estados activos en paralelo se pueden cargar más filas de bloques de memoria. Cada uno de estos super-estados cuenta con su propio contador de programa, que debe de tener soporte para la historia.

Los contadores se implementan como registros conectados a sumadores, tienen cada uno dos valores de referencia configurables y pueden ejecutar diferentes acciones. Las salidas y entradas están implementadas de forma similar a los contadores, pero las entradas son aún más sencillas, ya que no pueden ser modificadas por la statechart. La configuración se carga de forma serial, similar a como las FPGAs se configuran usando JTAG.

En cuanto a los recursos usados por esta implementación microprogramada de statecharts, es mucho mayor que la implementación en FPGAs escrita a mano, ya que puede ser uno o incluso dos órdenes de magnitud superior a ésta. Hay que mencionar sin embargo que esto incluye una cantidad de recursos que normalmente se reservan para actualizaciones futuras, y que mientras tanto se mantienen sin utilizar. Esto no suele ser un problema gracias al gran número de recursos que incluyen las FPGAs modernas. Además se observa que la implementación microprogramada tiene un alto coste fijo incluso para statecharts sencillas, y que la complejidad de los microprogramas crece más despacio que la complejidad de las statecharts.

Tener herramientas y procesos libres de errores es muy importante en grandes instalaciones como la European Spallation Source, ya que la complejidad del centro induce a la aparición de problemas que pueden impedir que la instalación funcione correctamente. Estos errores se pueden reducir utilizando herramientas y estrategias como las que presentamos en esta tesis. Además también hemos presentado contribuciones que hemos hecho para el sistema de temporización de la European Spallation Source, entre los

que se encuentran un nuevo modo de operación de los receptores de eventos, una nueva tarjeta para implementar receptores de eventos embebidos y una nueva herramienta para controlar la operación de la European Spallation Source. También se ha implementado la configuración del protocolo del sistema de temporización específicamente para el caso de la European Spallation Source. Los siguientes pasos en estos desarrollos son mejorar la herramienta que crea código VHDL a partir de statecharts e incluir en ella la metodología microprogramda que hemos propuesto.

Bibliography

[1] E. H. Lehmann, S. Hartmann, and M. O. Speidel, "Investigation of the content of ancient Tibetan metallic Buddha statues by means of neutron imaging methods," *Archaeometry*, vol. 52, no. 3, pp. 416–428, 2010.

[2] M. Munoz, "Description of modes for ESS accelerator operation." CHESS number: ESS-0038258.

[3] "Synchrotron radiation." `https://en.wikipedia.org/wiki/Synchrotron_radiation`. Accessed: 2019-08-15.

[4] ITER, "What is ITER?." `https://www.iter.org/proj/inafewlines`. Accessed: 2019-08-15.

[5] R. L. Cappelletti, C. J. Glinka, S. Krueger, R. A. Lindstrom, J. W. Lynn, H. J. Prask, E. Prince, J. J. Rush, J. M. Rowe, S. K. Satija, B. H. Toby, A. Tsai, and T. J. Udovic, "Materials research with neutrons at NIST," *Journal of Research of NIST*, vol. 106, pp. 187–230, jan-feb 2001.

[6] T. Panesor, "Neutron scattering," tech. rep., Institute of Physics Publications, 2011. `https://www.iop.org/publications/iop/2011/page_47521.html`.

[7] ILL, "What is the ILL." `https://www.ill.eu/about-ill/what-is-the-ill/`. Last accessed 2019-04-12.

[8] SNS, "Spallation Neutron Source." `https://neutrons.ornl.gov/sns`. Last accessed 2019-04-12.

[9] J-PARC, "What is J-PARC?." `http://j-parc.jp/public/en/about/about/index.html`. Last accessed 2019-04-12.

[10] ESS, "The European Spallation Source." `https://europeanspallationsource.se/about`. Last accessed 2019-04-12.

[11] R. Garoby, A. Vergara, H. Danared, I. Alonso, E. Bargallo, B. Cheymol, C. Darve, M. Eshraqi, H. Hassanzadegan, A. Jansson, I. Kittelmann, Y. Levinsen, M. Lindroos, C. Martins, Ø. Midttun, R. Miyamoto, S. Molloy, D. Phan, A. Ponton, E. Sargsyan, T. Shea, A. Sunesson, L. Tchelidze, C. Thomas, M. Jensen, W. Hees, P. Arnold, M. Juni-Ferreira, F. Jensen, A. Lundmark, D. McGinnis, N. Gazis, J. W. II, M. Anthony, E. Pitcher, L. Coney, M. Gohran, J. Haines, R. Linander, D. Lyngh, U. Oden, H. Carling, R. Andersson, S. Birch, J. Cereijo, T. Friedrich, T. Korhonen, E. Laface, M. Mansouri-Sharifabad, A. Monera-Martinez, A. Nordt, D. Paulic, D. Piso, S. Regnell, M. Zaera-Sanz, M. Aberg, K. Breimer, K. Batkov, Y. Lee, L. Zanini, M. Kickulies, Y. Bessler, J. Ringnér, J. Jurns, A. Sadeghzadeh, P. Nilsson, M. Olsson, J.-E. Presteng, H. Carlsson, A. Polato, J. Harborn, K. Sjgreen, G. Muhrer, and F. Sordo, "The European Spallation Source design," *Physica Scripta*, vol. 93, p. 014001, dec 2017.

[12] EPICS, "Experimental Physics and Industrial Control System." `https://epics-controls.org/`. Last accessed 2019-04-12.

[13] EtherCAT, "EtherCAT technology." `https://www.ethercat.org/en/technology.html`. Last accessed 2019-04-12.

[14] MTCA.4, "PICMG specification MTCA.4 revision 1.0." `https://www.picmg.org/openstandards/microtca/`. Last accessed 2019-04-12.

[15] A. W. Chao, K. H. Mess, M. Tigner, and F. Zimmermann, *Handbook of Accelerator Physics and Engineering*. World Scientific, 2nd ed., 2013.

[16] J. Serrano, P. Alvarez, M. Lipinski, and T. Wlostowski, "Accelerator timing system overview," in *Proceedings of 2011 Particle Accelerator Conference, New York, NY, USA*, pp. 1376–1380, 2011.

[17] The NTP Project, "The Network Time Protocol (NTP) distribution." `http://doc.ntp.org/3-5.93e/`. Accessed: 2019-09-04.

[18] IEEE Instrumentation and Measurement Society, *IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE, 2002. Sponsored by the Technical Committee on Sensor Technology (TC-9).

[19] T. Korhonen, "Review of accelerator timing systems," in *Proceedings of International Conference on Accelerator and Large Experimental Physics Control Systems, 1999, Trieste, Italy*, pp. 167–170, 1999.

[20] P. Moreira, J. Serrano, T. Wlostowski, P. Loschmidt, and G. Gaderer, "White Rabbit: Sub-nanosecond timing distribution over ethernet," in *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pp. 1–5, Oct 2009.

[21] ITU-T Study Group 15, *Timing and synchronization aspects in packet networks (Rec. ITU-T G.8261/Y.1361)*. International Telecommunications Union, 2008.

[22] MRF, "Micro-Research Finland Oy." `http://mrf.fi/`. Last accessed 2019-04-12.

[23] MRF, "MRF Open EVR." `https://github.com/jpietari/mrf-openevr`. Last accessed 2019-04-12.

[24] SLAC, "Linac Coherent Light Source." `https://lcls.slac.stanford.edu/`. Accessed: 2019-08-16.

[25] PSI, "Swiss Light Source - SLS." `https://www.psi.ch/en/sls`. Accessed: 2019-08-16.

[26] Diamond Light Source, "About us [Diamond Light Source]." `https://www.diamond.ac.uk/Home/About.html`. Accessed: 2019-08-16.

[27] Facility for Rare Isotope Beams, "About FRIB." `https://frib.msu.edu/about/index.html`. Accessed: 2019-08-16.

[28] A. X. Widmer and P. A. Franaszek, "A DC-balanced, partitioned-block, 8b/10b transmission code," *IBM Journal of Research and Development*, vol. 27, pp. 440–451, Sep. 1983.

[29] Xilinx, "Kintex-7." `https://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html`, 2010. Accessed: 2019-06-17.

[30] J. Martins, S. Farina, J. Lee, and D. Piso, "MicroTCA.4 Integration at ESS: From the Front-End Electronics to the EPICS OPI," in *Proc. of International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS'17), Barcelona, Spain, 8-13 October 2017*,

no. 16 in International Conference on Accelerator and Large Experimental Control Systems, (Geneva, Switzerland), pp. 1692–1694, JACoW, Jan. 2018. https://doi.org/10.18429/JACoW-ICALEPCS2017-THPHA133.

[31] S. Farina, J. Lee, J. Martins, and D. Piso, "MicroTCA Generic Data Acquisition Systems at ESS," in *Proc. of International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS'17), Barcelona, Spain, 8-13 October 2017*, no. 16 in International Conference on Accelerator and Large Experimental Control Systems, (Geneva, Switzerland), pp. 118–124, JACoW, Jan. 2018. https://doi.org/10.18429/JACoW-ICALEPCS2017-TUAPL01.

[32] Concurrent Techonologies, "Concurrent Techonologies." `https://www.gocct.com/`. Accessed: 2019-08-20.

[33] M. Davidsaver and EPICS community, "mrfioc2." `http://epics.sourceforge.net/mrfioc2/`. Last accessed 2019-04-12.

[34] J. Pietarinen, "MRF timing system." Timing Workshop at CERN Feb. 2008.

[35] G. Fatkin, Y. Macheret, A. Selivanov, A. Senchenko, and M. Vasilyev, "Modern Digital Synchronization Systems for Large Particle Accelerators," in *Proceedings, 26th Russian Particle Accelerator Conference (RuPAC 2018): IHEP, Protvino, Russia, October 1-5, 2018*, p. THCEMH02, 2018.

[36] T. Korhonen, "System requirement specification for the ESS timing system." CHESS number: ESS-0313920.

[37] J. Cereijo García, T. Korhonen, J. H. Lee, R. R. Osorio, and D. Piso Fernández, "Timing system at ESS," in *8th Int. Particle Accelerator Conf. (IPAC'17)*, 2017.

[38] Xilinx, "The Zynq-7000 SoC." `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`. Last accessed 2019-04-12.

[39] Arm, "Arm processors." `https://www.arm.com/products/silicon-ip-cpu`. Last accessed 2019-04-12.

[40] J. Jamroz, J. Cereijo García, T. Korhonen, and J. H. Lee, "Timing system integration with MTCA at ESS," in *17th Biennial International*

Conference on Accelerator and Large Experimental Physics Control Systems, 2019, New York, USA, 2019.

[41] J. Cereijo García, "Data model specification for the ESS timing system." CHESS number: ESS-0225673.

[42] J. Pietarinen, *Event System with Delay Compensation Technical Reference*. MRF.

[43] J. Cereijo García and R. R. Osorio, "Hardware implementation of statecharts for FPGA-based control in scientific facilities," in *XXXIV Conference on Design of Circuits and Integrated Systems, DCIS*, 2019.

[44] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[45] M. Milkes, "The genesis of microprogramming," *IEEE Annals of the History of Computing*, vol. 8, pp. 116–126, 1986.

[46] W. G. Spruth, *The Design of a Microprocessor*. Springer-Verlag, 1989.

[47] J. Cereijo García and R. R. Osorio, "A microprogrammed approach for implementing statecharts," in *Euromicro Conference on Digital System Design, 2019*, pp. 27–34, 2019.

[48] G. Booch, I. Jacobson, and J. Rumbaugh, "Unified Modeling Language." `http://www.uml.org/`, 1995. Accessed: 2019-06-017.

[49] D. Drusinsky and D. Harel, "Using statecharts for hardware description and synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 798–807, 1989.

[50] D. Drusinsky-Yoresh, "A state assignment procedure for single-block implementation of state charts," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 12, pp. 1569–1576, 1991.

[51] P. Clemente, P. Rundstadler, L. Specter, and K. Walsh, "From statecharts to hardware FPGA and ASIC synthesis," in *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, 1992.

[52] R. Kol, R. Ginosar, and G. Samuel, "Statechart methodology for the design, validation, and synthesis of large scale asynchronous systems,"

in *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 164–174, 1996.

[53] V. Salapura, G. Waleczek, and M. Gschwind, "A comparison of VHDL and statecharts-based modeling approaches," in *Proceeding of ITI*, 1994.

[54] V. Salapura and V. Hamann, "Implementing fuzzy control systems using VHDL and statecharts," in *EURO-DAC'96, European Design Automation Conference*, pp. 53–58, 10 1996.

[55] C. Veith, K. Buchenrieder, and A. Pyttel, "Mapping statechart models onto an FPGA-based ASIP architecture," in *EURO-DAC'96, European Design Automation Conference*, pp. 184–189, 1996.

[56] K. Buchenrieder and C. Veith, "A prototyping environment for control-oriented HW/SW systems using state-charts, activity-charts and FPGA's," in *EURO-DAC'94, European Design Automation Conference*, pp. 60–65, 1994.

[57] T. Muller-Wipperfurth and R. Hagelauer, "Graphical entry of FSMDs revisited: putting graphical models on a solid base," in *Proceedings Design, Automation and Test in Europe*, pp. 931–932, 1998.

[58] S. Qin and W.-N. Chin, "Mapping statecharts to Verilog for hardware/-software co-specification," in *FME 2003: Formal Methods*, pp. 282–300, 2003.

[59] S. Qin, W.-N. Chin, J. He, and Z. Qiu, "From statecharts to Verilog: a formal approach to hardware/software co-specification," *Innovations in Systems and Software Engineering*, vol. 2, pp. 17–38, Mar 2006.

[60] V.-A. V. Tran, S. Qin, and W. N. Chin, "An automatic mapping from statecharts to Verilog," in *Theoretical Aspects of Computing - ICTAC 2004*, pp. 187–203, 2005.

[61] R. Findenig, T. Leitner, V. Esen, and W. Ecker, "Consistent SystemC and VHDL code generation from state charts for virtual prototyping and RTL synthesis," in *Proceedings of DVCon*, 2011.

[62] Mathworks, "Stateflow HDL coder." `https://www.mathworks.com/products/hdl-coder.html`, 2018. Accessed: 2019-06-17.

[63] D.Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, and M. Trakhtenbrot, "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, pp. 403–414, apr 1990.

[64] D. Harel and H. Kugler, *The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)*, vol. 3147, pp. 325–354. Springer, 09 2004.

[65] D. Drusinsky, *Modeling and Verification Using UML Statecharts*. Newnes, 2006.

[66] Itemis, "Yakindu Statechart Tools." `https://www.itemis.com/en/yakindu/state-machine/`. Accessed: 2019-06-17.

[67] "Eclipse IDE." `https://www.eclipse.org/`. Accessed: 2017-09-05.

[68] H. Eeckhaut, "VHDL generation from Yakindu state charts with Xtend." `https://www.w3.org/WebPlatform/WG/`. Accessed: 2019-05-10.

[69] Apache Software Foundation, "Apache Xerces Project." `https://xerces.apache.org/`. Accessed: 2019-06-17.

[70] World Wide Web Consortium, "DOM, Document Object Model." `https://www.w3.org/TR/dom/`, 2015. Accessed: 2019-06-17.

[71] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Functional Optimization*. Springer, 1997.

[72] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Logic Optimization*. Springer, 1997.

[73] T. Ziadi, L. Helouet, and J. M. Jezequel, "Revisiting statechart synthesis with an algebraic approach," in *Proceedings. 26th International Conference on Software Engineering*, pp. 242–251, 2004.