Design of heterogeneous embedded systems in DFCharts

Ivan Radojevic

Abstract

System-level design has become a promising method of handling increasing complexity of embedded systems. Instead of making an early separation between software and hardware parts, system-level design starts at a higher level of abstraction where only the behaviour of a system is described without any implementation details. When the behaviour is captured with a formal model of computation, it becomes possible to automate verification and synthesis, which results in large productivity gains. Models of computation have been used successfully in design of control-dominated and data-dominated systems. However, there has been no satisfactory approach for modelling heterogeneous embedded systems, which contain both control-dominated and data-dominated parts. This thesis presents a model of computation for heterogeneous embedded systems called DFCharts. It targets heterogeneous systems by combining finite state machines (FSM) with synchronous dataflow graphs (SDFG). FSMs are connected in the same way as in Argos (a Statecharts variant with purely synchronous semantics) using three operators: synchronous parallel, refinement and hiding. The fourth operator, called asynchronous parallel, is introduced in DFCharts to connect FSMs with SDFGs. In the formal semantics of DFCharts, the operation of an SDFG is represented as an FSM. Using this representation, SDFGs are merged with FSMs so that the behaviour of a complete DFCharts specification can be expressed as a single, flat FSM. This allows system properties to be verified globally. The practical application of DFCharts has been demonstrated by linking it to widely used system-level languages Java, Esterel and SystemC. Furthermore, a design methodology based around DFCharts has been developed. It provides a complete design flow from DFCharts-based specification to implementation on multiprocessor architecture HiDRA. Due to the precise semantics of DFCharts, a large part of the design flow can be automated.
To my parents and brother
Acknowledgement

I would like to thank my supervisor Professor Zoran Salcic for his continuous support during the time I spent as a graduate student, from suggesting the first papers I should read to giving the final remarks on this thesis. His prompt replies to any issues I had helped me to proceed steadily with research. I would also like to thank my co-supervisor Dr. Partha Roop for his support. I highly appreciate his careful review of DFCharts semantics. Many valuable discussions I had with Professor Salcic and Dr. Roop were important for my progress.

I would like to thank graduate students from Embedded Systems and Control Systems research groups for many interesting conversations. Although everyone had a specific topic to work on, we could still help each other by discussing research problems in general.

Last but not least, I would like to thank my parents. Without their support, the demanding graduate study would definitely have been more difficult.
4.2.2 Data transfer from FSM to SDF ................................................................. 97
4.3 The impact of clock speeds ........................................................................... 98

**DFCharts in SystemC and Esterel** ................................................................. 101
5.1 Analysis based on requirements ................................................................... 102
  5.1.1 Concurrent processes ................................................................................ 102
  5.1.2 Rendezvous communication ..................................................................... 103
  5.1.3 Buffers and firing rules for dataflow ........................................................ 103
  5.1.4 HCFSM with synchronous/reactive communication .................................. 105
  5.1.5 Data transformations ................................................................................ 107
  5.1.6 Multiple processes inside a state ............................................................. 108
  5.1.7 Comparison between SystemC and Esterel ............................................. 110
5.2 Numerical results ........................................................................................... 110
5.3 Feature extensions of SystemC and Esterel .................................................. 114

**Java environment** .......................................................................................... 116
6.1 FSM classes ................................................................................................... 117
  6.1.1 Declaration of reference variables for I/O signals, states and variables ...... 118
  6.1.2 Inner classes for transition inputs and transition outputs ....................... 119
  6.1.3 Constructor parameters .......................................................................... 121
  6.1.4 Signal and shared variable connections, initialization of local variables ... 122
  6.1.5 Linking states, transition inputs and transition outputs ......................... 122
  6.1.6 Local signals, shared variables and channels for lower level FSMs and SDFGs ........................................................................................................ 123
  6.1.7 Instantiation of lower level FSMs and SDFGs ........................................... 126
  6.1.8 State refinement ...................................................................................... 126
6.2 SDFG Classes .................................................................................................. 127
  6.2.1 Constructor parameters ........................................................................... 128
  6.2.2 Instantiation of actors .............................................................................. 128
  6.2.3 Actor connections ................................................................................... 130
6.3 Top level classes .............................................................................................. 131
  6.3.1 Constructor parameters .......................................................................... 132
  6.3.2 Instantiation of input and output signals ................................................... 132
  6.3.3 Local signals, shared variables and channels for top level FSMs and SDFGs ........................................................................................................ 133
  6.3.4 Instantiation of top level FSMs and SDFGs .............................................. 133
  6.3.5 Top level refinement ............................................................................... 133
6.4 Simulation ....................................................................................................... 133
6.5 Library classes ............................................................................................... 134
  6.5.1 Base classes ............................................................................................ 135
  6.5.2 FSM component classes ......................................................................... 136
  6.5.3 FSM communication classes .................................................................... 137
  6.5.4 FSM-SDF communication classes ............................................................ 137
  6.5.5 Synchronization class ............................................................................. 138
6.6 Frequency relay in SystemC, Esterel and Java ............................................. 139
Implementation of DFCharts on HiDRA .............................................. 140
7.1 DFCharts design methodology ...................................................................... 141
  7.1.1 Specification ......................................................................................... 142
  7.1.2 FSM compositions .................................................................................. 142
  7.1.3 Allocation and partitioning ...................................................................... 143
  7.1.4 Synthesis ............................................................................................... 145
  7.1.5 Performance evaluation .......................................................................... 147
7.2 Execution of DFCharts specifications on HiDRA ............................................. 147
  7.2.1 Signals and variables ............................................................................... 147
  7.2.2 FSM thread ............................................................................................ 151
  7.2.3 Hierarchical and parallel compositions ................................................... 156
  7.2.4 FSM Scheduler ...................................................................................... 160
  7.2.5 Master tick handler ................................................................................. 161
  7.2.6 Slave tick handler .................................................................................. 165
7.3 Frequency relay implementation .................................................................... 166

Conclusion ........................................................................................................ 171
  8.1 Future research ......................................................................................... 174

References ........................................................................................................ 177
List of Figures

Figure 1.1: Graphical syntax of DFCharts ................................................................. 6
Figure 2.1: A Kahn process network example ........................................................... 10
Figure 2.2: An SDF graph ....................................................................................... 13
Figure 2.3: Esterel example ..................................................................................... 20
Figure 2.4: Argos example ...................................................................................... 23
Figure 2.5: Causality errors (a) no solution (b) multiple solutions ......................... 23
Figure 2.6: SystemC description of a counter ......................................................... 27
Figure 2.7: Java example ......................................................................................... 30
Figure 2.8: HiDRA configuration .............................................................................. 33
Figure 3.1: A DFCharts Specification ....................................................................... 36
Figure 3.2: Communication between an SDFG and a lower level FSM .................... 38
Figure 3.3: Transition priorities .............................................................................. 39
Figure 3.4: DFCharts specification with variables .................................................. 41
Figure 3.5: Communication between FSMs and SDFGs through variables .......... 43
Figure 3.6: Frequency relay – top level ................................................................. 45
Figure 3.7: Frequency and rate of change calculations ............................................ 47
Figure 3.8: Switch control and timer ..................................................................... 48
Figure 3.9: Parameter settings ............................................................................... 50
Figure 3.10: Threshold reception .......................................................................... 50
Figure 3.11: Possible extension of frequency relay ................................................. 51
Figure 3.12: Frequency relay in Ptolemy ............................................................... 53
Figure 4.1: Clocks in DFCharts ............................................................................. 58
Figure 4.2: Building the equivalent FSM for the example in Figure 3.1 .................... 60
Figure 4.3: Possible composition of a non-gclk FSM ............................................ 61
Figure 4.4: A multiclock FSM .............................................................................. 62
Figure 4.5: A multiclock synchronous system ....................................................... 63
Figure 4.6: A specification with one input signal and two conditions ..................... 66
Figure 4.7: A specification with two input signals and one channel ....................... 66
Figure 4.8: Synchronous product with a single clock ............................................ 68
Figure 4.9: Synchronous product with two synchronized clocks ............................ 68
Figure 4.10: Asynchronous product ..................................................................... 68
Figure 4.11: FSM A1 ............................................................................................. 71
Figure 4.12: FSM A2 ............................................................................................. 71
Figure 4.13: FSM A3=A1||A2 .............................................................................. 72
Figure 4.14: FSM A4 ............................................................................................. 76
Figure 4.15: FSM A5 ............................................................................................. 77
Figure 4.16: FSM A6 ............................................................................................. 78
Figure 4.17: FSM A7 .......................................................... 84
Figure 4.18: FSM A8 .......................................................... 84
Figure 4.19: FSM A9 = A7 ▼S12 A8 ................................................. 84
Figure 4.20: FSM A10 ................................................................ 85
Figure 4.21: FSM A11 = A9 ▼S11 A10 ........................................ 85
Figure 4.22: FSM that represents SDF graph with two inputs and two outputs .......... 90
Figure 4.23: Execution states of SDF1 from Fig. 3.5 .............................. 91
Figure 4.24: STF process ................................................................ 93
Figure 4.25: FTS process ................................................................ 94
Figure 4.26: A specification with behaviour sensitive to clock speeds .......... 99
Figure 5.1: Hierarchical FSMs in SystemC ........................................ 106
Figure 5.2: Esterel specification of timer in frequency relay .................. 107
Figure 5.3: Section of SystemC code for averaging filter in frequency relay .... 108
Figure 5.4: Modified model to better suite the SystemC specification ........ 109
Figure 6.1: Structure of FSM classes ............................................. 118
Figure 6.2: Signals, variables and states in FSM4 ................................ 118
Figure 6.3: The input for transition S41 → S42 in FSM4 ...................... 119
Figure 6.4: The output for transition S43 → S41 in FSM4 ...................... 120
Figure 6.5: The output for transition S42 → S43 in FSM4 ...................... 120
Figure 6.6: Parameters of FSM4 constructor ................................ 121
Figure 6.7: Connections and initializations in FSM4 ......................... 122
Figure 6.8: Creating transitions in FSM4 ........................................ 123
Figure 6.9: Signal start_roc, shared variable ave_freq and channel ch1 in FSM1 .... 124
Figure 6.10: The input for transition S31 → S32 in FSM3 ....................... 125
Figure 6.11: Instantiation of FSM3 ............................................... 126
Figure 6.12: Instantiation of FSM4 ............................................... 126
Figure 6.13: Instantiation of SDF1 .................................................. 126
Figure 6.14: Refinement of S2 in FSM1 ........................................ 127
Figure 6.15: Structure of SDF classes ............................................ 127
Figure 6.16: Class that specifies SDF1 ............................................. 128
Figure 6.17: Averaging filter actor in SDF1 ...................................... 129
Figure 6.18: Structure of top level classes ....................................... 131
Figure 6.19: Top level of frequency relay ........................................ 132
Figure 6.20: Execution of the top level class of the frequency relay .......... 133
Figure 6.21: Input file format ............................................................ 134
Figure 7.1: DFCharts based design flow ........................................ 141
Figure 7.2: Flow of control between FTs, FSM scheduler and tick handler .......... 146
Figure 7.3: Comparing memory and reactive signals .......................... 149
Figure 7.4: Architecture for DFCharts implementation ................................ 151
Figure 7.5: FSM thread .................................................................. 153
Figure 7.6: A DFCharts specification consisting of three FSMs .............. 157
Figure 7.7: Implementation of specification from Figure 7.6 without any FSM compositions ................................................................. 158
Figure 7.8: Implementation of specification from Figure 7.6 with hierarchical composition of FSM1 and FSM3 ........................................ 159
Figure 7.9: FSM scheduler ............................................................ 160
Figure 7.10: Master tick handler ...................................................... 162
Figure 7.11: Control of SDF iteration length ........................................ 163
Figure 7.12: DFCharts design flow without SDF iteration control................................. 164
Figure 7.13: Slave tick handler.................................................................................... 165
Figure 7.14: Partitioning for the second implementation option................................. 167
Figure 7.15: Frequency relay implementation with one ReMIC processor ............... 167
Figure 7.16: Frequency relay implementation with two ReMIC processors .......... 168
Chapter 1

Introduction

1.1 Embedded systems design

An embedded system represents a set of processing elements embedded inside a larger system. Although embedded systems are already widespread, the number of applications is expanding both in traditional areas such as communications, consumer electronics, aerospace, automotive, and in new ones such as biomedical.

Embedded systems differ in a number of ways from general purpose computing systems. An embedded system must be able to work at a speed imposed by the environment, which is not a requirement for a general purpose computing system. Concurrency and timing issues must be dealt with in embedded systems design. Furthermore, a range of constraints has to be faced. Some of the usual ones are performance, cost and power. While a general purpose computing system involves only software, an embedded system is often a mixture of software and hardware parts. Software runs on traditional microprocessors, digital signal processors (DSP) and various application specific processors (ASP). Hardware parts are implemented with application specific circuits (ASIC) or field programmable gate arrays (FPGA).
Traditionally, embedded system design starts with an informal specification. A decision is immediately made on how the functionality is going to be split between software and hardware. From this point, software and hardware parts are designed and verified separately. Software is written with programming languages like C/C++ and Java. Hardware description languages (HDL) like VHDL and Verilog are used for hardware design. The major difficulty comes when interfaces between software and hardware parts are created. At this stage many small implementation details must be handled. Many problems are discovered, which sometimes necessitate a complete redesign. Moreover, it is difficult to verify that the final implementation satisfies the original specification.

The traditional design methodology, which is still used in most embedded designs, may be satisfactory for smaller and medium sized systems. However, it is becoming increasingly difficult to use it for large systems. The functionality of embedded systems keeps growing. Technology advances make it possible to integrate an increasing number of components on a single chip. Moore’s law, which states that the number of transistors on a chip will double every two years, still holds. On the other hand, design methods are improving much slower. This results in a problem, known as productivity gap [1].

Design at higher levels of abstraction is a promising solution to overcome the productivity gap [2]. It should start with a specification created with a formal language. A variety of languages have been proposed for that purpose [3]. A specification should capture only the behaviour of a system without any reference to implementation. Various names have been used to refer to this level of abstraction such as system, behavioural, algorithmic or functional level. As much verification as possible should take place at the system-level. System-level verification is much faster than verification at lower levels of abstraction since many low level implementation details are not visible. Simulation is still the main verification method, but formal verification [4] is becoming important especially for safety critical embedded systems. After the verification has been completed, the final implementation consisting of software and hardware parts should ideally be synthesized automatically.
The behaviour of an embedded system is usually captured as a set of concurrent, communicating processes. The rules that are used for computation inside processes and communication between processes are determined by a model of computation [5],[6]. The computation inside each process is sequential. A finite state machine (FSM) [7] can be used to describe it, or more expressive models could be used that may have the full power of Turing machines. It is much more difficult to decide on the concurrency model that dictates the communication between processes. Currently, most embedded software design is done with threads which communicate using mutual exclusions locks, semaphores and interrupts. These communication mechanisms are difficult to use and often result in bugs that are hard to detect [8]. Sometimes a design may work for years before it suddenly crashes.

The alternative to threads are various concurrency models with formal semantics. They employ different scheduling policies and communication mechanisms resulting in different orderings of events. The most frequently used ones are discrete event [9], asynchronous dataflow [10], synchronous reactive [11], Petri nets [12] and process algebras such as communicating sequential processes (CSP) [13] and calculus of communicating systems (CCS) [14]. All these models impose some restrictions on communication but in return provide useful properties. The designer would have most freedom if communication between processes were unrestricted. However, tools would not be able to analyse such specifications and automated synthesis would not be possible. Instead, a specification would have to be refined manually to a lower level of abstraction. Because of manual refinement, it would be necessary to verify that the behaviour has been preserved.

In general there is a trade-off between analyzability and expressiveness in a model computation. A successful model needs to strike a balance. It needs to provide a framework for analysis but at the same time it needs to be expressive enough.

An important feature of embedded systems behaviour is heterogeneity. Two major types of embedded systems behaviour can be identified: control-dominated and data-dominated. Control-dominated systems have to quickly react to events that arrive from the external environment in irregular intervals. A lift controller and vehicle automation
are examples of control-dominated systems. Reaction time is less important for data-dominated systems which operate on samples that arrive in regular intervals. However, data-dominated systems perform computations that are much more intensive than those in control-dominated systems. Most of digital signal processing algorithms, such as FFT, FIR and IIR filters are data-dominated. Most embedded systems contain both control-dominated and data-dominated behaviours. For example, a mobile phone has to perform data-dominated operations such as speech processing, coding, modulation but it also has to take care of control-dominated operations such as network protocols or reacting to user commands.

With the integration of analogue parts together with digital parts on a single chip, heterogeneity of embedded systems will become even more pronounced. Models that aim to address analogue/digital systems need to be able to support continuous time. In this thesis, we are concerned with purely digital systems. Thus when we refer to heterogeneous embedded systems we mean systems that represent a mixture of data-dominated and control dominated parts.

Well established models of computation have specific advantages but are not able to successfully handle complete heterogeneous embedded systems. Asynchronous dataflow models [10] consist of processes that communicate through FIFO channels with blocking reads. Using blocking reads ensures that outputs do not depend on the scheduling order and speeds of processes. Dataflow models have been successfully applied in signal processing and other transformational systems. However, they lack reactivity because of blocking reads. In synchronous/reactive model [11], all processes read inputs and produce outputs simultaneously when a clock tick occurs. Blocking reads are not necessary for determinism. This feature is evident in synchronous language Esterel [15], which is deterministic, but has plenty of reactive statements. Synchronous/reactive model has also had success in the signal processing area with languages Lustre [16] and Signal [17], which have dataflow flavour. The requirement that all events in a system are synchronous can have a big implementation price. This especially applies in the design of large systems on chip. It may mean that all processing elements in a system must wait for each other at some point even though
they are performing unrelated tasks. In the context of pure hardware implementation, distributing a single clock can be a problem.

In this thesis we present DFCharts, a model of computation that targets heterogeneous embedded systems by combining dataflow and synchronous/reactive models. We show its practical use by linking it to selected system level languages and implementation architecture.

Modelling heterogeneous embedded systems is still an open research area with plenty of room for advancement. This is witnessed by the absence of mature, well-developed tools for heterogeneous systems. Esterel studio [18] is a commercial design environment from Esterel technologies, which uses Esterel language for creating specifications. It is very convenient for design of control-dominated systems but lacks features to support data-dominated behaviour. Another tool from Esterel technologies called SCADE is based around Lustre, but it also allows insertion of FSMs to support control-dominated behaviour. The combination of Lustre and FSMs is still completely synchronous. Hence it could have implementation difficulties when applied in design of large embedded systems. In order to produce efficient implementations, Polis [19] and its commercial successor VCC use a globally synchronous locally asynchronous (GALS) model called Codesign finite state machines (CFSM). However, their target is just control-dominated behaviour. Ptolemy [20] is a graphical environment that supports a variety of models of computation. It is still primarily used for simulation. Automatic synthesis has been demonstrated for synchronous dataflow (SDF) domain, but not for heterogeneous specifications. Simulink [21], while it does have some code generation capabilities, is also primarily a simulation environment.

1.2 DFCharts

DFCharts combines hierarchical concurrent finite state machines (HCFSM) [22] with synchronous dataflow graphs (SDFG) [23]. Three communication mechanisms are employed in DFCharts: synchronous broadcast used between FSMs, FIFO channels used inside SDFGs, rendezvous channels used between FSMs and SDFGs. The
semantics of synchronous broadcast is as in Argos [24], a Statechart variant with purely synchronous communication. An SDFG can be placed anywhere in the FSM hierarchy. Thus, it is possible to specify complex control logic that leads to activation and termination of SDFGs. When expressed with its graphical syntax, the DFCharts model looks as shown in Figure 1.1.

We have described the semantics of communication between FSMs and SDFGs in detail within tagged-signal model (TSM) framework [25]. Another type of semantics we use is based on automata. It is similar to the semantics of Argos. It represents the operation of an SDFG as an FSM. In this way a complete DFCharts model can be flattened to produce a single FSM. This in turn allows the model behaviour to be analysed globally. The automata semantics can handle any dataflow model that has a clearly defined iteration and executes in bounded memory. For example, cyclo-static dataflow (CSDF) [26] can be easily incorporated in DFCharts, but Kahn process network (KPN) [27] would require an extension in the current semantics. In this thesis, we focus on SDF, which is suitable for a vast range of applications.
All FSMs in a DFCharts model are driven by a single clock. On the other hand, each SDFG operates at its own speed. Structurally, a DFCharts model resembles an Esterel program with asynchronous tasks where asynchronous tasks are comparable to SDFGs and possibly other dataflow models placed in DFCharts. However, asynchronous tasks in Esterel are essentially part of the external environment, outside the Esterel semantics as we discussed in [28]. They are very vaguely defined. In DFCharts semantics, SDFGs are fully integrated.

The contributions of the thesis can be summarized as follows:

- A model of computation with formal semantics has been developed. It targets heterogeneous embedded systems which involve control-dominated and data-dominated parts.
- Although DFCharts primarily serves as a model of computation that is used by system-level languages for specification of heterogeneous systems, it can be used directly for specification. For that purpose, graphical syntax has been developed.
- The application of SystemC and Esterel in specifying heterogeneous embedded systems has been analysed using DFCharts. Possible improvements have been suggested for both languages.
- A simulation environment for DFCharts based designs has been created in Java. The main feature is a class library that enables straightforward specification of FSMs, SDFGs and their communication in Java.
- A design methodology for heterogeneous embedded systems has been proposed. It gives an implementation path from DFCharts models to multiprocessor architecture HiDRA. Mapping and execution of DFCharts on HiDRA is presented in detail.

1.3 Thesis organization

The rest of the thesis is structured as follows. Chapter 2 gives background material for the chapters that follow. Chapter 3 describes DFCharts using its graphical syntax and
compares it with related models. It also shows how DFCharts can be applied in modelling a practical heterogeneous embedded system called frequency relay. This case study is also used in all other chapters. Chapter 4 gives the formal semantics of DFCharts. Chapter 5 deals with specification in SystemC and Esterel. Chapter 6 describes Java environment for simulating DFCharts designs. Chapter 7 describes the design flow from DFCharts to HiDRA. Chapter 8 concludes the thesis and discusses directions for further research.
Chapter 2

Background

This chapter looks at several topics that are relevant for the following chapters. Sections 2.1 and 2.2 describe two models of computation that are integrated in DFCharts, asynchronous dataflow and synchronous reactive. They are relevant in all chapters. The next four sections deal with languages. Sections 2.3 and 2.4 describe synchronous languages Esterel and Argos. Esterel appears in Chapter 5 while Argos is important in Chapters 3 and 4 which describe syntax and semantics of DFCharts. Sections 2.5 and 2.6 describe object oriented languages SystemC and Java. They appear in Chapters 5 and 6 respectively. Finally, section 2.7 describes multiprocessor architecture HiDRA. It is relevant for implementation of DFCharts, described in Chapter 7.

2.1 Dataflow models

Synchronous dataflow (SDF), which is the main focus of this section, belongs to the group of dataflow models in which processes communicate through first-in-first-out (FIFO) channels using blocking reads. In order to understand the properties of SDF, we need to explore the most general model in the dataflow group, called Kahn Process Networks (KPN) [27].
KPN processes communicate through FIFO channels using blocking reads and non-blocking writes. At any point during the execution of a Kahn process network, a process can either be waiting for an input or doing computations. Since blocking reads are used, a process cannot test a channel for the presence of data. If a process attempts to read from an empty channel it will become blocked until data arrives on the channel.

Figure 2.1 shows a KPN example that was used in [27]. The two instances of process $h$ copy data from input to output, but initially they output 0 and 1. Without these initial tokens the network would be deadlocked from the start. Process $g$ copies data from input channel $X$ to output channels $T_1$ and $T_2$ alternately. Process $f$ copies data alternately from input channels $Y$ and $Z$ to output channel $X$.

![Figure 2.1: A Kahn process network example](image)

In the denotational semantics of Kahn process networks, processes are mathematically defined as functions that map potentially infinite input streams to output streams. A stream is a sequence of data elements $X = [x_1, x_2, x_3, x_4, \ldots]$. The empty sequence is marked by the symbol $\bot$. A relation on streams called prefix ordering [10] is useful for analyzing the mathematical properties of Kahn process networks. For example the sequence $X = [0]$ is a prefix of the sequence $Y = [0,1]$ which is in turn a prefix of $Z = [0,1,2]$. The relation “$X$ is a prefix of $Y$ or equal to $Y$” is written as $X \sqsubseteq Y$.

A chain is an ordered set in which any two elements are comparable. Alternate names for a chain are linearly ordered set and totally ordered set [29]. In the context of Kahn
process networks, elements of a chain are sequences and they are compared with the prefix ordering relation \( \subseteq \). Any increasing chain \( \vec{X} = (X_1, X_2, \ldots) \) with \( X_1 \subseteq X_2 \subseteq \ldots \) has a least upper bound \( \Pi \vec{X} \) (symbol \( \Pi \) is used to denote a least upper bound). A least upper bound is a sequence whose length tends towards infinity:

\[
\lim_{i \to \infty} X_i = \Pi \vec{X}
\]

The set of all sequences is a complete partial order (c.p.o.) with the relation \( \subseteq \), since any increasing chain of sequences in this set has a least upper bound.

In Kahn process networks, a process \( f \) maps input streams to output streams. A process \( f \) is continuous if and only if for any increasing chain \( \vec{X} = (X_1, X_2, \ldots) \):

\[
f(\Pi \vec{X}) = \Pi f(\vec{X})
\]

If a process is continuous, it is also monotonic (the opposite is not necessarily true). Monotonicity means that:

\[
X \subseteq Y \implies f(X) \subseteq f(Y)
\]

A process network can be described with a set of equations, with one equation for each process. For example, the process network in Figure 2.1 can be represented by the following set of equations:

\[
T_1 = g_1(X), T_2 = g_2(X), X = f(Y, Z), Y = h(T_1, 0), Z = h(T_2, 1)
\]

The system of equations above can be reduced to a single equation. For instance the equation for \( X \) is:

\[
X = f( h(g_1(X), 0), h(g_2(X), 1) )
\]
If all processes are continuous the set of equations has a unique least fixpoint solution. The solution represents the histories of tokens that appeared on the communication channels. For example, the solution for X is an infinite sequence of alternating 0's and 1's - \( X = f(Y,Z) = [0,1,0,1 \ldots] \). The proof by induction can be found in [27].

The blocking read semantics of Kahn process networks ensures that processes are continuous. Therefore a set of equations describing a Kahn process network will have a unique least fixed point solution. This leads to a very useful property of KPN. Any execution order of processes will yield the same solution i.e. the same histories of tokens on the communication channels.

While the execution order cannot influence the histories of tokens, it can greatly impact memory requirements (buffer sizes). Since writes are non-blocking, there are no restrictions on buffer sizes. There are two major methods for scheduling Kahn process networks, data-driven scheduling and demand driven scheduling [30]. In data driven scheduling, the semantics of the Kahn process networks is satisfied in a simple way - a process is unblocked as soon as data is available. Data driven scheduling can lead to unbounded accumulation of tokens on communication channels.

An alternative strategy is to use demand driven scheduling of processes where a process is activated only when the tokens it produces are needed by another process. Kahn and MacQueen describe a demand driven scheduling method in [31]. A process that needs tokens is marked as hungry which causes the producer of those tokens to be activated. That in turn can cause another activation. All the scheduling is done by a single process.

Regardless of the type of scheduling employed, decisions in KPN have to be made at run time. Thus, context switching becomes inevitable if multiple processes run on a single processor. Valuable time has to be spent on saving the state of the current thread before the control can be transferred to another thread.

Synchronous dataflow (SDF) [23], [32] imposes limitations on KPN in order to make static scheduling possible. An SDF network is composed of actors that are connected by FIFO channels. When an actor fires, it consumes tokens from input channels and
produces tokens on output channels. Firings of an SDF actor create a process. The firing rule of an actor specifies how many tokens are consumed on each input. In SDF, the constant number of tokens is consumed on each input in every firing i.e. the firing rule remains the same. It should also be emphasised that an SDF actor has to output a constant number of tokens on each output in every firing. Due to constant consumption and production rates of tokens it is possible to make very efficient static schedules.

Figure 2.2 shows an SDF network that consists of three actors. Consumption and production rates are labelled on each channel. For example, from the direction of the ch1, it can be seen that its production rate is RA1 and its consumption rate is RC1.

![Figure 2.2: An SDF graph](image)

There are two steps in constructing a static schedule for an SDF graph. The first step is to determine how many times each actor should fire during an iteration. An iteration is a series of actor firings that return the channels to their original state. The first step is accomplished by solving the set of balance equations [32]. Balance equations state that production and consumption of tokens must be equal on all channels. The balance equations for the SDF graph from Figure 2.2 are shown below.

\[
FA \times RA1 = FC \times RC1 \\
FA \times RA2 = FB \times RB1 \\
FB \times RB2 = FC \times RC2
\]

FA, FB, FC are integers showing how many times actors A, B and C fire in a single iteration. They form a firing or repetition vector. The least positive integer solution is
taken. For example if $RA1 = 2$, $RA2 = 2$, $RB1 = 3$, $RB2 = 3$, $RC1 = 6$ and $RC2 = 6$ then $FA = 3$, $FB = 2$ and $FC = 1$.

If the only solution to the set of equations is zeros, the SDF graph is said to be inconsistent [33]. This means that production and consumption of tokens cannot be balanced on all channels. As a result, the execution of an inconsistent SDF graph results in unbounded accumulation of tokens on channels. The graph from Figure 2.2 would become inconsistent if $RC1$ were equal to 5, for example.

The second step is to analyse data dependencies between SDF actors in order to determine the order of firings. Multiple valid execution orders can emerge from the analysis due to different interleaving of actor firings. For example, the execution orders AAABBC and AABABC can both be used for the SDF graph from Figure 2.2.

An SDF graph can contain a cycle that is formed by two or more channels. Initial tokens must be placed on a channel in the cycle so that deadlock does not occur. This was relevant for the KPN example from Figure 2.1 where the two $h$ processes produced initial values.

SDF is suitable for a wide range of DSP systems with constant data rates. It has efficient static scheduling and always executes in bounded memory. These properties are very useful in embedded systems design. For this reason, SDF graphs have been adopted as a part of DFCharts. For systems with variable rates, KPN or dynamic dataflow models can be used. There are many dataflow models whose expressiveness falls between SDF and KPN such as boolean dataflow (BDF) [34] cyclostatic dataflow (CSDF) [26], parameterized synchronous dataflow (PSDF) [35], multidimensional synchronous dataflow [36], synchronous piggyback dataflow [37] among others.

A large amount of research has been done on synchronous dataflow resulting in numerous techniques and algorithms for memory optimization [38],[39], [40], [41], [42], [43], [44]; simulation [45], [46]; software synthesis [47], [48] hardware synthesis [49], [50], [51]; HW/SW codesign [52], [53], [54].
2.2 Synchronous/reactive model

The synchronous reactive (SR) model of computation [11] is the underlying model for the group of synchronous languages which includes Esterel [15], Argos[24], Lustre[16] and Signal [17]. A brief description of all four languages can be found in [55]. In the SR model of computation time is divided into discrete instants. In each instant (tick), inputs are read and outputs are computed instantaneously. This is the central assumption in the synchrony hypothesis of the SR model. Instantaneous computation and communication makes outputs synchronous to inputs. The status of each signal has to be defined in each tick. It can be either present (true) or absent (false).

This model is similar to synchronous digital circuits that are driven by clocks. As a result, SR models can be efficiently synthesised into hardware. Software synthesis is also possible. In that case, the time between two successive instants is usually not constant.

The assumption of instantaneous computation facilitates hierarchical specification of systems. When a process is broken down into several other processes, they will all have instantaneous computation.

Zero delay communication represents a challenge for compilers of synchronous languages. An SR compiler has to be able to deal with causality loops that arise as a result of zero delays. When resolving the status of each signal in a tick three general outcomes are possible:

- There is a single solution. The signal is either present or absent.
- There is no solution. The model does not make sense.
- Both the presence and absence of the signal satisfy the model. Thus, the system is non-deterministic.
The first outcome is the desired one. The last two outcomes should be rejected by the compiler and an error should be reported to the user. The three possible cases are illustrated in sections 2.3 and 2.4 with Esterel and Argos programs.

There are two distinct styles of synchronous modelling [11], which emerged during the development of the synchronous languages. The first one is known as State Based Formalisms, the second one is known as Multiple Clocked Recurrent Systems (MCRS's). The oldest and most developed synchronous language Esterel, uses the first style. Argos is also an SBF-style synchronous language. On the other hand, declarative dataflow languages Lustre and Signal use the second style.

State based formalisms are convenient for specifying control-dominated systems but they are not efficient in dataflow modelling. It is the opposite with MCRS's. Their main use is in specifying signal processing systems but it is more difficult to specify systems that step through different states. There have been attempts to unify two styles in a single environment as in [56], [57].

Synchronous programs can always be compiled into finite state machines. This property is very important since it greatly facilitates formal verification and ensures that memory requirements are known at compile time.

In Kahn process networks and related dataflow models, events are partially ordered. Events on a single channel are totally ordered, but they in general have no relation with events on other channels. In synchronous models, events are totally ordered. This has an important impact in modelling reactive systems, which have to promptly respond to every event that comes from the external environment. They often have to wait for several events simultaneously. A KPN process cannot wait on multiple channels at the same time since it must implement blocking reads in order to achieve determinism. On the other hand, a synchronous process can test a channel before reading it and still preserve determinism. The downside of the total ordering of events is that it may unnecessarily reduce the implementation space by overspecifying the system, especially in the case of data-dominated-systems. Furthermore, distributed implementation with synchronous models is much more difficult than with dataflow models.
It is interesting to note that due to the differences in event ordering, synchronous dataflow (SDF) is not an appropriate name when KPN based dataflow models are compared against synchronous models. To avoid confusion, a better name would be statically scheduled dataflow (SSDF) as suggested in [6].

### 2.3 Esterel

Esterel is an imperative language based on the synchronous reactive model of computation that was described in section 2.2. Since the synchrony hypothesis is used in Esterel, it is assumed that all communications and computations take zero time. As a result outputs are synchronous to inputs. Communication is achieved by synchronous broadcast. Events produced by one process are immediately visible to other processes. Esterel is a deterministic language. For a given sequence of inputs only one sequence of outputs is possible.

Esterel programs communicate with external environment through signals and sensors. Signals are used as inputs and outputs. Sensors can only be used as inputs. Signals can be *pure* or *valued*. Pure signals only have status at each instant. The status of a signal indicates whether the signal is present or absent. Valued signals also carry a value besides status. Valued signal S is denoted as S(v) where v represents the value. In expressions the value of the signal S is denoted as ?S. If different values of the same signal are produced by multiple processes at the same time instant, it has to be specified how those values are combined. For example different values can be added to produce the resulting value. It can also be specified that it is forbidden to have multiple values for a signal at any time instant.

There are two types of Esterel statements: *primitive* statements and *derived* statements. Derived statements are actually derived from primitive statements. From the point of view of the programmer, derived statements are user-friendly and they also make programs shorter. An example of how a statement is derived will be shown later in this section. Primitive statements can be divided in two groups: basic imperative statements and temporal statements. Basic imperative statements take no time, they are executed
instantaneously. Temporal statements handle signals and they take time. Table 2.1 lists the primitive statements, which form the basic Esterel kernel. The table was taken from [15] slightly modified to reflect recent changes in the Esterel syntax.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>nothing</td>
<td>dummy statement</td>
</tr>
<tr>
<td>halt</td>
<td>halting statement</td>
</tr>
<tr>
<td>X := exp</td>
<td>assignment statement</td>
</tr>
<tr>
<td>call P (variable-list) (expression-list)</td>
<td>external procedure call</td>
</tr>
<tr>
<td>emit S(exp)</td>
<td>signal emission</td>
</tr>
<tr>
<td>stat1; stat2</td>
<td>sequence</td>
</tr>
<tr>
<td>loop stat end</td>
<td>infinite loop</td>
</tr>
<tr>
<td>if exp than stat1 else stat2 end</td>
<td>conditional</td>
</tr>
<tr>
<td>abort stat when S</td>
<td>watchdog</td>
</tr>
<tr>
<td>stat1</td>
<td></td>
</tr>
<tr>
<td>trap T in stat end</td>
<td>trap definition</td>
</tr>
<tr>
<td>exit T</td>
<td>exit from trap</td>
</tr>
<tr>
<td>var X : type in stat end</td>
<td>local variable declaration</td>
</tr>
<tr>
<td>signal S (combine type with comb) in stat end</td>
<td>local signal declaration</td>
</tr>
</tbody>
</table>

Table 2.1: Esterel basic kernel

Most of the statements in Table 2.1 are found in other imperative languages. The statements emit and abort when are specific to Esterel. Nothing does nothing and takes no time. Halt does nothing but it never terminates so it takes time. The assignment statement and external procedure call are both instantaneous. Emit evaluates the value of the signal, emits the signal with its value and terminates. The emission is instantaneous. The sequence of statements is executed sequentially but since the computation is assumed to be infinitely fast the whole sequence takes no time. For example, a sequence

\[ X := 1; X := X + 1 \]
results in X being equal to 2. The sequence above takes zero time, but it is executed in the correct order. The number of statements in a sequence has to be finite. For example statements such as

\[ X := 0; \text{loop} \ X := X + 1 \text{ end} \]

are not allowed. The loop construct never terminates. When the body of a loop terminates the loop is instantly restarted. The expression in \textit{if then else} conditional can be composed of both signals and variables in the latest version of Esterel. Thus, the \textit{present} statement which is used exclusively for testing signals became obsolete.

The execution of \textit{abort when} can end in two ways. The body inside the construct can finish before signal S occurs. If S occurs before the body finishes, the body is terminated instantly without being allowed to execute in the instant of termination. This type of abort is \textit{strong}. It is also possible to create a \textit{weak} abort simply by writing \textit{weak abort} instead of \textit{abort}. Weak abort can also be made using \textit{trap}. When weak abort is used the body is allowed to execute in the instant of termination.

The branches of a parallel statement start simultaneously. A parallel statement terminates when all of its branches have terminated.

The \textit{trap exit} construct is a powerful control mechanism also found in some other languages. \textit{Trap} defines a block of statements that instantly terminate when the corresponding \textit{exit} statement is executed.

There are several useful derived statements in Esterel such as \textit{await}, \textit{every}, \textit{each}, \textit{sustain}. It is shown below what \textit{await} is equivalent to.

\[ \text{await } S \quad \text{instead of} \quad \text{abort} \]
\[ \quad \text{halt} \]
\[ \quad \text{when } S \]
The basic unit of an Esterel program is called *module*. A module consists of the *declaration part* and *statement part*. In the declaration part, types, constants, functions and procedures used in the module are declared. Esterel is used in combination with a host language, which can be C for instance. Earlier Esterel versions had only basic types and operators built in such as integer, Boolean, basic arithmetic and logic operators. More complex operators and types had to be imported from the host language. The latest Esterel version [58] includes more complicated data types that are mainly geared towards hardware modelling. The declaration part can also specify relations between signals. For example a relation can state that two signals never occur at the same time. This helps the Esterel compiler in optimisation. An example of an Esterel module is given in Figure 2.3.

```esterel
module est_example:
    input A, B, R;
    output O : integer;

    loop
        abort
            [ await A || await B ];
        emit O(1)
        when R
    end
    ||
    loop
        await C;
        emit R
    end

end module
```

Figure 2.3: Esterel example

The first top level parallel branch emits O with the value of 1 after either A or B occurs. This operation can be pre-empted by R which is emitted by the second parallel branch when C occurs.

It was mentioned in section 2.2 that the powerful zero delay communication mechanism of the SR model can create causality problems. In particular two kinds of problems were mentioned: lack of solution and multiple solutions. Those two problems will now be
illustrated on Esterel programs. The program below demonstrates the first problem – no
solution.

signal S in
    present S else emit S end
end

If it is assumed that signal S is not present then it is emitted so it is present. This
contradiction cannot be resolved since the whole statement is executed in a single
instant. The second program demonstrates the second causality problem – multiple
solutions.

signal S1, S2 in
    present S1 else emit S2 end
    ||
    present S2 else emit S1 end
end

The program has two solutions. In one solution S1 is present and S2 is absent. In the
other solution S2 is present and S1 is absent. Hence the program is nondeterministic. An
Esterel compiler has to reject both programs.

The first Esterel compiler [15] converts an Esterel program into a flat FSM. The
resulting code is very fast but it is impractical for large systems due to exponential
increase in code size. The second Esterel compiler [59] translates an Esterel program
into a synchronous digital circuit. The resulting code is compact, but its execution is
very slow since every gate has to be evaluated in every reaction. Recent Esterel
compilers [60], [61], [62] produce fast and compact code by simulating the reactive
features of Esterel but they cannot compile a number of correct cyclic Esterel programs.

Esterel’s reactive statements have inspired several modifications of C/C++ and Java to
make them more suitable for embedded systems. Some examples are SystemJ[63], ECL
[64], Jester[65], Reactive-C[66], JavaTime [67].
2.4 Argos

A single, flat FSM is suitable for specifying sequential controllers, but it can hardly handle more complicated embedded systems where support for concurrency and hierarchy becomes necessary. This weakness was addressed in Statecharts [22] which introduced hierarchical and concurrent compositions of FSMs and several other useful features. It was described in [22] that concurrent FSMs communicate using synchronous broadcast with the assumption of instantaneous communication. However, many questions regarding instantaneous loops were left unanswered. This semantic gap resulted in more than twenty different versions of Statecharts [68] approaching causality problems in different ways. Argos is a version of Statecharts that uses the principles of Esterel to deal with semantic challenges caused by instantaneous communication. Unlike other Statecharts variants, Argos compiler rejects all programs that contain non-determinism that the programmer did not intend to have i.e. implicit non-determinism. Another distinguishing feature of Argos is that it forbids inter-level transitions, which connect two states on different hierarchical levels. This emphasizes modularity.

The three basic Argos operators that are applied on FSMs are: synchronous parallel, refinement and hiding (localization). They were introduced in [69]. Later, a few additional operators were described in [24]. Figure 2.4 shows an Argos specification with the three main operators.

The top level FSMs are connected by the synchronous parallel operator and communicate using the local signal \( c \). When FSM1 is in S11 and \( a \) is present, it makes the transition to S12 and emits \( c \). If FSM2 is in S22, the presence of \( c \) instantaneously triggers the transition to S21. When FSM1 makes the transition from S12 to S11 it preempts FSM3. Only weak abort is available in Argos. Thus FSM3 is allowed to react in the tick in which it is pre-empted. For example if it is in S32 and \( e \) is present in the instant of pre-emption, \( r \) will be emitted.
As in Esterel, causality problems due to instantaneous communication can also appear in Argos. Two examples are shown in Figure 2.5. A dot between input signals means logical AND (conjunction). A line above the name of a signal denotes its absence.

The specification in Figure 2.5 (a) has no meaningful behaviour. If \( c \) is present and \( a \) is absent, FSM1 makes the transition from \( S_{11} \) to \( S_{12} \) and emits \( b \). Since \( b \) is present
FSM2 makes the transition from S21 to S22 and emits a. This means that a is present and absent simultaneously, which is not possible. Figure 2.5 (b) illustrates non-determinism. If c is present, both FSMs can take transitions and emit signals a and b that are necessary for their activations or transitions are not taken and a and b are not emitted.

2.5 SystemC

SystemC [70], [71], [72] is an attempt to expand the widely used programming language C++ with features that would make it suitable for specifying embedded systems. SystemC is a class library of C++ and can be compiled on any C++ compiler. This is a clear advantage since there are well-developed, mature tools for C++. SystemC includes constructs to support specification of embedded systems characteristics such as timing, concurrency and reactive behaviour. Such construct cannot be found in the standard C/C++ programming environment.

SystemC enables designers to represent both hardware and software parts of an embedded system in the single environment. A specification in SystemC is executable because it can be simulated. In fact SystemC allows specifying and simulating a system at different levels of abstraction ranging from highly abstract system models down to cycle based models. The ability to simulate a high level system specification brings large benefits to the design process. System functionality can be verified before implementation begins. Mistakes can be uncovered early in the design process when it is much cheaper to remove them than in well advanced design stages.

The main idea behind SystemC version 1.0 is to enable the designers to describe both software and hardware using the same language. For that purpose some features of hardware description languages were incorporated in the SystemC class library. Later SystemC 1.0 evolved into SystemC 2.0. The current version is 2.2 but we will focus only on major changes that occurred between the releases 1.0 and 2.0. With respect to SystemC 1.0, SystemC 2.0 is better equipped for system-level modelling. This is mainly due to new communication mechanisms that appeared in SystemC 2.0. The second
version still supports everything from the first version. In the following paragraphs SystemC 1.0 is firstly described. Later in the section, the features that defined SystemC 2.0 are outlined.

A specification in SystemC consists of modules. A module can instantiate lower level modules thus supporting hierarchy. Modules contain processes that describe system functionality. Three kinds of processes exist as will be described below. Modules are connected through ports which can be bi-directional or uni-directional. Ports are connected with signals. There is a wide range of signal types available in SystemC in order to support different levels of abstraction. Clocks are also available as a special signal type.

Breaking a system into modules enables division of tasks among designers. A module can be modified such that its external interface and functionality remain the same and the only thing that is changed is the way in which the function is described. In this way other modules in the design are unaffected. The external interface of a module is represented by its ports. There are three types of ports: input, output and input - output ports. Every port also has a data type associated with it. Modules are declared with the SystemC keyword SC_MODULE. For example the declaration of a module that describes a fifo (first-in-first-out) buffer is given below.

```systemc
SC_MODULE (fifo) {
    sc_in <bool>    load;
    sc_in <bool>    read;
    sc_inout <int>  data;
    sc_out <bool>   full;
    sc_out <bool>   empty;

    // module description not shown
}
```

Only the ports of the module are shown not the functionality. The module has two input ports, two output ports and one bi-directional input-output port. The input and output
ports are used for control and are all of type boolean. The bi-directional data port is of type integer.

Ports of different modules are connected with signals. A signal declaration only indicates which data type is carried by the signal, the direction (in, out, inout) is not specified as in the case of ports. The direction of data flow in a signal is determined by ports that are connected together by the signal.

Local variables can be used to store data received through ports. Local variables are only visible inside the module in which they are declared unless they are explicitly made to be visible in other modules. Local variables can be of any C++ data type, SystemC data type or user-defined data type.

The functionality in a module is defined by processes. Processes are registered with the SystemC kernel. Processes are sensitive to signal changes. A process starts executing when there is a change in at least one of the signals the process is sensitive to. Some processes are similar to C++ functions. They execute sequentially and return control once they have stepped through all statements. Other processes are different in that they may be suspended by halting statements and then resumed at some instant.

There are three types of processes in SystemC: methods, threads and clocked threads. Methods behave like C++ functions. A method starts executing statements sequentially when called. It returns control to the calling mechanism when it reaches the end. The execution of a method has to be completed. For that reason it is recommended in SystemC User's Guide that designers should be careful to avoid making infinite loops inside a method.

Threads may be suspended and resumed. A thread is suspended when the wait( ) function is encountered. It is resumed when one of the signals in its sensitivity list changes.

Clocked threads are a special case of threads. Clocked threads are suspended and resumed in the same way as threads. However the sensitivity list of a clocked thread
contains only one signal and that signal is a clock. Furthermore a clocked thread is sensitive to only one edge of the clock. Clocked threads resemble the way in which synchronous digital circuits are specified. For example a synchronous process in VHDL contains only a clock signal in its sensitivity list.

The code in Figure 2.6 illustrates how a counter is specified in SystemC inside a module.

```cpp
#include "systemc.h"

SC_MODULE (count) {
    sc_in <bool>     load;
    sc_in <int>        din;
    sc_in <bool>     clock;
    sc_out <int>      dout;

    int count_val;       // internal data storage

    void count_up( );

    SC_CTOR (count)    {
        SC_METHOD (count_up);              // Method process
        sensitive_pos   <<  clock;
    }

    void count : : count_up ( ) {
        if (load) {
            count_val = din;
        } else {
            count_val = count_val +1;
        }
        dout = count_val;
    }

    Figure 2.6: SystemC description of a counter

    The behaviour of the module is described by the process count_up which is of the type method. The process is triggered by the positive edge of the clock. This is specified in the line "sensitive_pos << clock". When the process is triggered the value on input port load is checked. If it is true variable count_val is assigned the value of input port din. Otherwise count_val is incremented by one. count_val is a local variable, visible only in
the module. It should be noted that every module is initialized by a constructor. The keyword SC\_CTOR is used for constructors.

In SystemC 1.0 modules communicate through ports which are connected by hardware signals. The behaviour of those signals is essentially the same as in VHDL and Verilog. From the point of view of the system level designer, communication using only hardware signals is insufficient to capture highly abstract features that are apparent at the system level. For that reason new communication mechanisms were added leading to the second release. For example in SystemC 2.0, a semaphore or a mutex are available to protect shared data used by communicating modules. Modules can also communicate using FIFOs, another type of communication not directly supported in SystemC 1.0. Moreover, it is possible for users to define their own communication mechanisms. In order to do that the user has to define a new channel and its interfaces. Channels and interfaces are two new kinds of objects in SystemC 2.0. A channel defines a communication mechanism. An interface is used to connect a port of a module to a channel. FIFO, mutex and semaphore are examples of built-in SystemC 2.0 channels.

With abstract communication mechanisms, it becomes possible to work at levels of abstraction that are higher than RTL. Transaction level modelling (TLM) [71] is an example of such level. In TLM, processing elements typically communicate by procedure calls whereby a whole packet of data can be transferred. Implementation details seen at the register transfer level are omitted. As a result, TLM simulations are much faster than RTL simulations. Extensive research in modelling embedded systems at the TLM level has been carried out recently [73], [74], [75], [76], [77], [78].

Designs in SystemC can be simulated with testbenches. A testbench typically contains a module that generates inputs for the design under test (DUT), the DUT itself, and a module that checks the outputs of the DUT. It is often the case that a testbench does not contain a module for checking outputs; instead the designer manually checks outputs. Designs at various levels of abstractions can be simulated.

It is worth mentioning that the development of interfaces and channels in SystemC 2.0 was significantly influenced by another system-level language based on C, called SpecC.
[79]. While SystemC libraries are defined using the standard C/C++ syntax, SpecC extends the standard ANSI-C by adding new constructs to support hardware and system-level features. HardwareC [80] and Handel-C [81] are also examples of languages that extend C in order to support hardware design.

2.6 Java

Java [82] was proposed as a platform independent object oriented language for embedded devices. One of the most important characteristics of Java is that it is not intended to be compiled into executable object code. The product of Java compilation is normally byte code, which is not processor dependent. In order to run byte code, a CPU has to use Java Virtual Machine (JVM). The purpose of JVM is to interpret the instructions contained in byte code.

Although initially intended for embedded systems, the greatest success of Java came with the emergence of the Internet where portability became essential. However, the feature of Java that probably contributed the most to its popularity in Internet applications is security. It does not allow malicious programs that can gain unauthorized access to important data.

Java inherits most of its syntax from C. It shares many principles of object oriented programming with C++, including the three basic ones: polymorphism, encapsulation and inheritance. However, Java does not have upward or downward compatibility with C/C++. It is also not expected to replace C++. While it is more suitable than C++ for certain application domains, it runs several times slower since it is not directly compiled into object code.

Object oriented programming is emphasized even more in Java than C++, since all data and methods must be contained inside classes. Figure 2.7 shows a simple program with two classes.
The first class specifies a rectangle. Its data consist of two variables of primitive type double that represent the height and the width of a rectangle. It has two methods that are used for specifying the length of the sides and displaying the area.

Besides the two methods, there is a constructor. A constructor has to have the same name as the class it belongs to. It is invoked only once when an instance of the class is created. For this reason, the method setSides exists in class Rectangle. Although it has exactly the same functionality as the constructor it can be invoked at any time during the lifetime of an object.

Destructors, which appear in C++, do not exist in Java. Thus, there is no need for the delete operator. When an object goes out of scope, it is automatically collected by Java runtime environment.
The second class is used for executing the first one. When a Java application starts running, the execution begins in a main method. When there are several classes that contain main methods, the programmer has to specify the starting point. In Java, a main method has to belong to a class like all other methods. In C++, functions can be outside classes.

Initially a single application thread is running, but it is possible to spawn other threads. Java provides facilities that support multi-threading. On the other hand, C++ is not capable of supporting multi-threading directly. Instead, multiple threads can only be created through an operating system.

The first line of the main method in class RectangleDemo instantiates an object of type Rectangle using the new operator. At the same time, arguments are passed to the constructor of Rectangle. The variable named rectangle is the reference to the object. The only way to access an object in Java is to obtain its reference. In contrast, C++ allows both pass by reference and pass by value. A pass by reference can be done with reference variables, but it can also be simulated with pointers. In Java, pointers do not exist. This is perhaps the most important difference between Java and C++. Pointers were excluded from Java for two reasons. The first is security risks. Pointers can be used by malicious programs to gain access to memory addresses that are outside the Java run-time system. The second is the difficulty in using pointers that often results in programming mistakes.

In the rest of the main method, the area of the rectangle is displayed for two different lengths of the sides. It can be seen that class members are accessed with a dot.

Java is an attractive language for embedded systems because of its security, portability and object oriented programming. The large footprint of Java runtime environment and its slow execution due to the interpretation of byte codes are not important problems for enterprise applications. However, these disadvantages must be addresses in embedded applications. For that purpose, various processors that are optimized for Java execution have emerged. PicoJava [83] is probably the most popular, but many others have been
used such as [84], [85], [86]. Compilation techniques aimed at faster execution have also been proposed [87], [88], [89].

2.7 HiDRA

HiDRA [90] is a multiprocessor architecture which targets heterogeneous embedded systems by providing a variety of features for implementation of control-dominated and data-dominated behaviours. The main building block of HiDRA is ReMIC [91], a processor with special instructions that support reactivity. Besides ReMIC, it is possible to include application – specific processors (ASP). Furthermore, HiDRA supports software-hardware codesign by incorporating functional units that are implemented in hardware.

A distinguished processor in HiDRA is master processor. It is used for system initialization and synchronization of concurrent activities. The other processors in the architecture are called slave processors. Together with functional units, they implement application behaviours. It should be noted that the master processor can implement application behaviours as well.

Processors perform communication and synchronization using signals and shared memories. Signals are implemented with memory-mapped registers or reactive signals of ReMIC. It is possible to create various communication mechanisms such as rendezvous, FIFO, synchronous broadcast. EMPEROR [92], [93], an architecture which is similar to HiDRA but more restricted, focuses on the implementation of Esterel’s communication mechanisms.

An example of HiDRA configuration is given in Figure 2.8. Besides the master processor, it contains two slave processors and two functional units.
ReMIC belongs to the family of reactive processors which also includes ReFLIX [94] and RePIC [95]. The reactivity features are implemented in a similar way in all three processors, but ReMIC has a larger instruction size and more developed datapath than the other two processors. It employs Harvard architecture where program and data buses are separated. The width of the program memory is 32 bits while the width of the data memory is 16 bits.

ReMIC consists of three units: processor control unit, reactive functional unit (RFU) and RISC-type pipelined datapath. RFU is responsible for the implementation of reactivity instructions, which are listed in Table 2.2. They enable efficient compilation of control-dominated languages by directly supporting features such as signal emission, preemption, delay, waiting for signal, checking signal presence.

<table>
<thead>
<tr>
<th>Features</th>
<th>Instruction syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal emission</td>
<td>EMIT signal(s)</td>
</tr>
<tr>
<td>Signal sustenance</td>
<td>SUSTAIN signal(s)</td>
</tr>
<tr>
<td>Signal polling</td>
<td>SAWAIT signal</td>
</tr>
<tr>
<td>Delay</td>
<td>TAWAIT clock(s), prescaler</td>
</tr>
<tr>
<td>Conditional signal polling</td>
<td>CAWAIT signal1, signal2, address</td>
</tr>
<tr>
<td>Signal presence</td>
<td>PRESENT signal, address</td>
</tr>
<tr>
<td>Preemption</td>
<td>ABORT signal, address</td>
</tr>
</tbody>
</table>

Table 2.2 Reactivity instructions in ReMIC
Chapter 3

Specification in DFCharts

This chapter describes how embedded systems are specified in DFCharts. Section 3.1 presents an introduction to specification in DFCharts. Section 3.2 illustrates the application of DFCharts on a practical heterogeneous embedded system called frequency relay. Section 3.3 discusses languages and models related to DFCharts.

3.1 Introduction to DFCharts

DFCharts targets heterogeneous embedded systems by mixing finite state machines (FSM) with synchronous dataflow graphs (SDFG). All FSMs in a DFCharts specification are driven by the same clock. On the other hand, each SDFG operates independently at its own speed. A DFCharts specification typically consists of multiple levels of hierarchy. The highest hierarchical level is referred to as “top level”. FSMs and SDFGs that are placed at the top level are always active. Lower hierarchical levels are obtained by refining states of FSMs. When a state is entered, the objects that refine it are activated. They are terminated when the state is left.
3.1.1 Operators

The three most important operators in Argos (synchronous parallel, localization, and refinement) are also used in DFCharts to combine FSMs. The scope of the refinement operator is extended in DFCharts, since a state of an FSM can be refined by an SDFG. There are three ways in which a state of an FSM can be refined: only by FSMs; only by SDFGs; by FSMs and SDFGs at the same time. The fourth operator of DFCharts, named asynchronous parallel, is used to connect an FSM and an SDFG. The use of the operators is illustrated in Figure 3.1, which shows a DFCharts specification. It consists of seven FSMs and two SDFGs with three hierarchical levels.

The two top level FSMs, FSM1 and FSM2, execute concurrently as indicated by the synchronous parallel operator. In each tick of the clock, they read inputs and instantaneously produce outputs. Local signal $d$ is used for synchronization. When FSM1 is in state S13 and input $b$ is present, the transition to S11 is taken and $d$ is emitted. If FSM2 is in state S21, $d$ causes the transition from S21 to S22.

When state S22 of FSM2 is entered, FSM3 is started. When $e$ is present, FSM2 leaves S22 and FSM3 is terminated in the same instant. Weak aborts are used in DFCharts as in Argos, which means that an FSM is allowed to produce outputs in the instant it is aborted. Thus, if both $e$ and $g$ are present, and FSM3 is in state S31, then $t$ is emitted. Another type of refinement is illustrated in FSM1, where state S12 is refined by one SDFG and two FSMs.

An SDFG has two kinds of buffers: internal and interface. Internal buffers are used for communication between actors within a graph. Interface buffers, which can be input or output, are used for communication between actors and FSMs or external environment. The execution of an SDFG consists of a series of iterations. Before each iteration the SDFG has to receive inputs and send outputs produced during the previous iteration. As no outputs have been computed before the first iteration, the output buffers must contain initial tokens. Initial tokens can also be placed on internal buffers to prevent deadlock.
Figure 3.1: A DFCharts Specification
Input and output buffers are connected to the external environment and FSMs through channels. In Figure 3.1, $ch3$ and $ch4$ connect SDF1 to the external environment while $ch1$ and $ch2$ connect it to FSM4 and FSM5. The direction of each channel is indicated by an arrow. The communication between an SDFG and an FSM through a channel can occur only when both sides are ready for it. When the communication on a channel has taken place, both sides are notified by the event called rendezvous. It should be mentioned that in CSP [13], processes also have to meet in order to communicate. However, the procedure leading to rendezvous is slightly different. In CSP, both reads and writes are blocking. If the sender wants to send, but the receiver is not ready to receive, the sender will block. Similarly, if the receiver is ready to receive, but the sender is not ready to send, the receiver will block. In DFCharts, an SDFG cannot start the next iteration until communications on all of its channels have been completed. However, an FSM can abort waiting for rendezvous regardless of whether it is attempting to send or receive.

As mentioned previously, an SDFG is ready to communicate on all channels when it finishes an iteration. An FSM is ready to communicate on a single channel when it enters a rendezvous state. A state is called rendezvous state if it has an outgoing transition that is triggered by rendezvous. Rendezvous states cannot be refined. An FSM may spend many ticks in the rendezvous state waiting for rendezvous, but rendezvous itself is instantaneous – the communication on a channel happens in a single tick. In Figure 3.1, the rendezvous states are S41, S52 and S71.

When FSM5 is in S52, it is ready to send on $ch2$ as seen from the transition $ch2!$. The CSP notation is used where ‘?’ is used to denote input while ‘!’ is used to denote output. When SDF1 is ready to receive on $ch2$, the rendezvous occurs triggering the transition $ch2!$, which leads to S51. In this example, $ch2$ is used purely for synchronization. SDF1 only receives a token that acknowledges that the rendezvous has been completed on the channel. There is no real data flowing through. More interesting examples with typed channels will be presented later in this section after the introduction of variables.

For semantic purposes, we will assume that the communication between an SDFG and external environment also occurs when both sides are ready. However, in practical
embedded applications, the external environment cannot wait. Data will be lost if the SDFG is not ready to receive it in its input buffers. Consequentially, there is a timing constraint that has to be satisfied. This issue will be dealt with in Chapter 7, which discusses implementation of DFCharts.

It should be noted that after each iteration of an SDFG, the rendezvous on each channel occurs only once. Suppose that FSM4 enters S42 in the instant when the rendezvous on ch1 occurs. In the next tick of the FSM clock, j is present which causes FSM4 to go to S41 again. In the meantime SDF1 has not started a new iteration since the rendezvous on ch2 has not occurred yet. The rendezvous will not occur again on ch1 until SDF1 completes a new iteration.

An FSM can be at a lower hierarchical level than an SDFG it communicates with. An example of that situation is given in Figure 3.2. SDF1, placed at the top level communicates on ch2 with FSM2 that refines state S12. This example also shows that waiting for rendezvous can be pre-empted by a higher level transition; in this case triggered by input signal a.
3.1.2 Transition priorities

In Figure 3.1, state S61 has an outgoing transition labelled $q \land \lnot p$ where $\land$ denotes logical $\text{and}$ while the bar above $p$ denotes $\text{not}$. Besides $\text{and}$ and $\text{not}$ the logical connective $\text{or}$ ($\lor$) can also be used in construction of Boolean expressions. The purpose of the $\text{and}$ connective in this example is to make $p$ and $q \land \lnot p$ exclusive so that the execution of the FSM is deterministic. The transition triggered by $p$ has the priority over the one triggered by $q \land \lnot p$. Making triggers exclusive in this way can be difficult if a large number of signals are involved. In DFCharts, this problem can be handled by specifying transition priorities directly. This is illustrated in Figure 3.3 by adding priority labels on transitions.

A rendezvous state can be left due to transitions that are triggered by signals. However, the transition that is triggered by rendezvous must always have the highest priority as shown in state S31. A rendezvous must happen if it is enabled.

A rendezvous state can be left due to transitions that are triggered by signals. However, the transition that is triggered by rendezvous must always have the highest priority as shown in state S31. A rendezvous must happen if it is enabled.

![Figure 3.3: Transition priorities](image)

3.1.3 Variables

The examples presented so far did not contain any data. Few practical applications can be specified by FSMs that only use pure signals. For this reason, variables can be employed in DFCharts. An FSM that uses variables becomes FSMD (FSM with datapath). Variables are also important for communication between FSMs and SDFGs.
Under rare circumstances will an SDFG just synchronize with one or more FSMs, which was the case in Figure 3.1. Usually, it will send and receive data.

In DFCharts, variables can be shared or local. A local variable can only be used by a single FSM while shared variables can be used by multiple FSMs. However, among concurrently running FSMs, only a single FSM may write to a shared variable. The value of a local variable can change during a tick. On the other hand, a shared variable cannot have its value changed during a tick. If the writer updates a shared variable, the new value will be visible to the readers only in the next tick.

When variables are used, transitions can also contain conditions on variables and procedures that update variables. The general form of a transition is \( t[c]/O,P \) where \( t \) is the transition trigger (either Boolean expression on signals or rendezvous), \( c \) is the condition on variables, \( O \) is the set of emitted output signals and \( P \) is the set of invoked procedures. Note that a transition cannot have a condition on variables if it is triggered by rendezvous.

Figure 3.4 shows a DFCharts specification with variables. Each variable has to be declared, and initialized if necessary. The declaration has to indicate the data type of the variable. In the Java environment implementing DFCharts, described in Chapter 6, only primitive data types float, integer and Boolean are currently supported for shared variables. However, the type of a local variable may be any data structure created in Java.

FSM1 contains two variables, local variable \( lv \) and shared variable \( sv \). \( lv \) is only seen by FSM1 while \( sv \) can also be accessed by other FSMs that declare it. \( sv \) is written in the procedure that is called when FSM1 makes the transition from S11 to S12. In that procedure, \( lv \) is first incremented by one, and then \( sv \) gets the new value of \( lv \) multiplied by two. Since the value of a shared variable remains constant in a single tick, a procedure like \( \{ sv=sv+1; sv=sv*2; \} \) would be pointless. The first statement would not have any effect on \( sv \). The new value seen in the next tick will be determined by the last assignment, which is \( sv=sv*2 \) in this case. Smaller procedures, like the one just described, can be immediately shown in curly brackets. The names of the larger
procedures such as proc1 and proc2 are given in italics and the contents are defined elsewhere.

Figure 3.4: DFCharts specification with variables

FSM2 reads sv in conditions sv<18 and sv>11. Similarly to procedures, short conditions are immediately shown, while long ones like c1 and c2 in FSM6 are indicated in italics and defined elsewhere. Since FSM2 is just a reader of sv, all descendants from FSM2 would only be able to read sv as well. Descendants from FSM1 may also write sv but in any single tick only one FSM may be the writer. FSM3 and FSM5, which write sv in
procedures \textit{proc1} and \textit{proc2} respectively, are both allowed to be writers since they cannot be active at the same time. On the other hand, FSM3 and FSM4 cannot both write. FSM4, like FSM6, can only read $sv$.

It should be pointed out that outgoing transitions of a hierarchical state that is refined by an FSM that writes a shared variable cannot contain procedures that write the same shared variable. This can be seen in Figure 3.4. The transitions going out of S12 and S13 do not contain procedures that write $sv$, since FSM3 and FSM5 write $sv$. The purpose of this restriction is to prevent FSMs at different hierarchical levels to write the same shared variable simultaneously.

\section*{3.1.4 Data transfer between FSM and SDF}

The condition that there may be only one active writer of a shared variable can easily be abandoned if resolution functions are introduced, as in VHDL for example. In fact, the behaviour of variables shared between FSMs is not an essential part of DFCharts semantics. A lot more important aspect, which is characteristic to DFCharts, is how variables are used to enable transfer of data between FSMs and SDFGs. This is where array variables become important. Figure 3.5 shows a DFCharts specification, which illustrates how variables are used in communication between FSMs and SDFGs.

When channels are used for data transfer, not just for mere synchronization as in Figure 3.1, their data type must be declared. As in the case of shared variables, the DFCharts Java environment currently supports only the transfer of primitive data types across channels. Besides the data type, the declaration of a channel also indicates how many tokens pass through it when rendezvous occurs. In Figure 3.5, $ch2$ transfers two tokens, all the others transfer only one.

When rendezvous occurs on $ch1$, FSM1 makes a transition from S11 to S12, but at the same time, the received integer is stored in shared variable $x$. FSM2 sends data from variable $y$ on $ch2$, which is stored in the input buffer of actor A, which belongs to SDF1. The number next to the actor indicates that two tokens are stored. Thus, FSM2 has to send two integers. This in turn means that variable $y$ has to be an array having two
integers, as seen in its declaration. The elements of $y$ are accessed using the usual array notation, which can be seen from the procedure called when FSM2 makes the transition of priority 1 from S1 to S2. Variables needed for transmission or reception of multiple tokens, such as $y$, must be arrays and not simply sets of elements where the order does not matter. The reason is that the order in which tokens are stored in SDF buffers is often significant.

Figure 3.5: Communication between FSMs and SDFGs through variables

FSM3 communicates with SDF2 and SDF3 using local variable $z$. In S31, it is ready to send the value contained in $z$ on $ch_6$ to SDF2. In S32, it is ready to receive a value on
ch7 from SDF3 and store it in z. In the instant when rendezvous occurs, an implicit procedure occurs that copies the value from a variable to a channel or vice versa. Instead of ch7!z we could write ch7/{z = ch7_token} where ch7_token denotes the value that is received on ch7. Similarly, instead of ch6!z we could write ch6/ {ch6_token = z}. The transition from S32 to S31 also calls a procedure that multiplies z by two. It must be emphasized that the implicit rendezvous procedure is always executed first. The variable is first modified by rendezvous and then multiplied by two. It cannot be the other way around. Otherwise, non-determinism could occur due to different execution orders.

When an FSM uses a shared variable for rendezvous, it must be the only active writer of the variable. This ensures that the value of the variable will not change while the FSM is in the rendezvous state.

### 3.2 Case Study: Frequency Relay

Power systems need protection from overloading. When a power system is overloaded some loads must be disconnected in order to prevent damage. A significant decrease in the frequency level of the main AC signal whose normal value is 50 (or 60) Hz indicates that the system may be dangerously overloaded. The same problem is also detected when the rate of change of the AC signal is too fast. The frequency relay is a system that measures the frequency and its rate of change in a power network. Measurement results are compared against a set of thresholds that can be modified through a user interface. There are six thresholds in total, three for the frequency and three for the rate of change. If the current thresholds indicate that the frequency is too low or its rate of change too fast some loads are disconnected from the network by opening one or more switches (three in the presented case), as determined by a decision algorithm. Loads are gradually reconnected if the frequency and its rate of change improve. The specification consists of seven FSMs and one SDFG. In the following figures, we will show all signals and channels but only important variables.
Figure 3.6 shows the top level FSM. The main operation in the system occurs in S12 which is entered from the initial state S11 when on is present. In that transition, the thresholds are set to default values in \textit{init\_thresh()} procedure. S31, where nothing happens, is entered from S12 when off is present. S11 can be reached from the other two states with reset.

3.2.1 Peak detection

S12 is refined by six FSMs and one SDFG. The purpose of SDF1 is to find the time between every two consecutive peaks in the AC waveform. With this information, the frequency can easily be determined. SDF1 consists of three blocks, each having a single
input and a single output. All consumption and production rates are equal to one token. Thus, SDF1 is a homogeneous graph.

The AC signal is sampled by an analogue-to-digital (ADC) converter, which is not a part of the specification since DFCharts handles purely digital systems. The sampling frequency is 8 kHz. Samples are sent on ch1 to SDF1. They first go through the averaging filter, which is designed to remove some noise from the signal. After the averaging filter, the signal is processed by the symmetry function block. The algorithm performed in this block may be expressed by the following equation:

\[ r(x) = \sum_{\theta = 0}^{\pi/2} (L + f(x + \theta))(L + f(x - \theta)) \]  
(3.1)

where \( L \) is a positive constant, \( f(x) \) the input signal and \( r(x) \) symmetry function that indicates maxima of the input function. It can be noticed from the equation above that the algorithm resembles autocorrelation. It magnifies the maximum points in the signal thus making their detection in the presence of noise easier. Maximum points are used as reference points for the frequency calculation. More details on the theory behind the signal processing operations performed inside the symmetry detection block can be found in [96]. Using the results from the symmetry detection block, the peak detection block identifies peaks in the waveform. A sample is a peak if it is larger than its predecessor and it successor. When a sample is a peak, the peak detection block sends on ch1 the number of samples counted between the current peak and the previous one. Otherwise, zero is sent.

3.2.2 Frequency and rate of change calculations

The data sent on ch1 is received by FSM3, shown in Figure 3.7. In the instant when rendezvous takes place on ch1, FSM3 makes a transition from S31 to S32 while storing the number of samples in local variable \( din \). In the next tick, the value of \( din \) is examined. If it is zero, the transition is taken back to S31, where a new rendezvous is awaited. Otherwise, the transition to S33 is taken, which calls procedure \( af\_calc() \) (average frequency calculation). The instantaneous frequency is easily calculated by dividing the sampling frequency with the number of samples between the last two
peaks. In order to smooth out any spikes in the AC waveform, the frequency is averaged over four values. In the next tick, the average frequency value is compared with the three frequency thresholds (thr1, thr2 and thr3) in procedure \( fs\_calc() \) (frequency status calculation). \( Fs \) (frequency status) is an integer that indicates the position of the average frequency with respect to the thresholds. It can take four values: 0 if \( \text{ave}_\text{freq} > \text{thr}_1 \), 1 if \( \text{thr}_1 > \text{ave}_\text{freq} > \text{thr}_2 \), 2 if \( \text{thr}_2 > \text{ave}_\text{freq} > \text{thr}_3 \), and 3 if \( \text{thr}_3 > \text{ave}_\text{freq} \).

**FSM4** – rate of change calculation  
**FSM3** – frequency calculation

<table>
<thead>
<tr>
<th>variable: float ave_freq=0; int fs=0, rs=0;</th>
</tr>
</thead>
</table>
| S41 \( \text{start}_\text{roc/}
| \text{aroc}\_\text{calc()} \) \( \text{true/}
| \text{rd} \) |
| S42 \( \text{true/}
| \text{rs}\_\text{calc()} \) |
| S43 |

<table>
<thead>
<tr>
<th>[( \text{din}==0 )]</th>
</tr>
</thead>
<tbody>
<tr>
<td>S31 ( \text{ch1}?\text{din} )</td>
</tr>
</tbody>
</table>
| S32 \( [\text{din}!==0] \)
| \( \text{/af}\_\text{calc()} \) |
| S33 \( \text{true/}
| \text{fs}\_\text{calc()},\text{start}_\text{roc} \) |
| S34 |

| signal: start_roc, rd; |

Figure 3.7: Frequency and rate of change calculations

The rate of change of the frequency is handled by FSM4, shown in Figure 3.7. It remains in the initial state S41 until FSM3 emits \( \text{start}_\text{roc} \). When \( \text{start}_\text{roc} \) is present the transition from S41 to S42 is made calling \( \text{aroc}\_\text{calc()} \) (average rate of change calculation) procedure. The instantaneous rate of change is calculated in the procedure from the current and previous values of the average frequency. The average rate of change is then calculated using four values. In the next tick, \( \text{rs}\_\text{calc()} \) is called, which determines the value of \( rs \) (rate of change status) based on the average rate of change and the three related thresholds \( th4, th5 \) and \( th6 \). The thresholds are used in exactly the same way as with \( fs \). Thus, \( rs \) also takes one of the possible four values, 0, 1, 2 or 3. Finally, the transition from S43 to S41 emits \( rd \) (roc done) which brings FSM3 to the initial state and tells FSM5 to check the values of \( fs \) and \( rs \).
3.2.3 Switch control

FSM5, shown in Figure 3.8, determines how many switches should be closed (turned on) using the values of $fs$ and $rs$. In fact, each state of FSM5 directly corresponds to the state of the switches. In S51 all three switches are closed; in S52 two are closed; in S53 one is closed; all three are open in S54. The presence of $rd$ signals that the values of $fs$ and $rs$ have been updated leading to a possible state change. The values of $fs$ and $rs$ are read in conditions $c1$, $c2$, and $c3$, which stand for $fs = 1 \lor rs = 1$, $fs = 2 \lor rs = 2$, and $fs = 3 \lor rs = 3$ respectively. Increasing values of $fs$ and $rs$ indicate increasing network load. FSM5 immediately has to respond by opening an appropriate number of switches. On the other hand, when the network condition begins to improve, which is marked by decreasing values of $fs$ and $rs$, switches are reconnected one by one after a certain period of time. This is why there is a transition from S51 to S54, but not the other way around. To get from S54 to S51, FSM5 must go through S53 and S52.

Figure 3.8: Switch control and timer
The time it takes for a switch to get reconnected represents a number of ticks, counted by FSM6 (Figure 3.8). FSM5 and FSM6 communicate through the shared variable of type Boolean $to$ (time out) and local signal $st$ (start timer). All transitions in FSM5 start the timer except the one from S52 to S51. That transition does not start the timer since there are no more switches left to reconnect. When the timer is started, FSM6 makes the transition from S61 to S62 and sets the count variable $v$ to zero. In each tick $v$ is incremented until it reaches the limit. This process can be reset any time by another $st$. When the limit is reached FSM6 goes back to S61 with $to$ becoming true. A transition can then be triggered in FSM5 with the condition $== T$.

### 3.2.4 Threshold modification

Threshold modification is handled by FSM2 and FSM7, shown in Figures 3.9 and 3.10. The process of entering new thresholds begins when input $sth$ (start thresholds) is present. FSM2 emits $inth$ (input threshold) and enters state S22, which is refined by FSM7. Thresholds are received one by one in FSM7. In each transition $nt$ (next threshold) is emitted, except for the transitions from S76 to S77, which emit $alld$ (all done). Each threshold has two possible values that are selected by inputs $thresh0$ and $thresh1$. It is also possible to leave a threshold unchanged using $skip$. The number of possible values for each threshold can be increased by using binary coding so that two inputs can cover four values. Moreover, the number of inputs could be increased. The selection is recorded with procedure $set_ft$ for frequency thresholds and $set_rt$ for rate of change thresholds.

In FSM7 new threshold values are just recorded and do not take effect until procedure $update$ in FSM2 is completed in the transition from S23 to S21. It is possible to pre-empt FSM7 either by $done$ or $cancel$. When $done$ occurs all new threshold values recorded before the pre-emption are written in $update$. In contrast, when $cancel$ occurs all recorded values are cleared in procedure $clr$. 
3.2.5 Optional CDMA link

For some applications, it may be desirable that the system is capable of receiving the thresholds from a remote site. For that purpose, a CDMA link could be added, which is
partly shown in Figure 3.11. This is only an optional part of the system and does not appear in Chapters 5, 6 and 7, which use the case study. It is shown here to additionally demonstrate the practical use of DFCharts modelling.

The SDF graphs represent the CDMA transmitter and receiver, which operate as described in [97]. In a single iteration the transmitter encodes a frame of 80 information bits to produce an outgoing packet. The receiver decodes an incoming packet to produce a frame of 80 information bits. Note that modulation, demodulation and RF stages are not part of the model. FSM2 handles the protocol for receiving frequency and rate-of-change thresholds. The frame of 80 information bits it sends to the transmitter on \( ch3 \) contains an instruction requesting a threshold to be entered. The frame of 80 information bits it reads from the receiver through \( ch4 \) contains a threshold or an input command.

---

### SDF2 - transmitter

- **CRC encoder**
- **Add 8-bit encoder tail**
- **Convolutional encoder**
- **Symbol repetition**
- **Block interleaver**

**ch3**

**FSM2**

### SDF3 - receiver

- **CRC decoder**
- **Remove repeated symbols**
- **Block deinterleaver**
- **Remove 8-bit encoder tail**
- **Convolutional decoder**

**ch4**

**modulation and RF**

**demodulation and RF**

---

*Figure 3.11: Possible extension of frequency relay*
3.3 Related languages and models

There have been a variety of approaches in dealing with behavioural heterogeneity of embedded systems. A popular one is to extend Kahn process networks. The pure KPN model is non-reactive since blocking reads are used to insure determinism. Yapi [98] is a model which extends KPN by allowing a channel to be tested for the presence of data i.e. non-blocking read is used to make selected channels reactive. Non-blocking reads are also used in reactive process networks (RPN) [99] to enable reactivity. In both models determinism is inevitably lost. Also, even though reactivity increases their expressiveness they are still not as good for control dominated systems as synchronous reactive models.

Heterogeneity is addressed in Funstate [100] by placing in parallel an HCFSM model with communication as in Argos and a dataflow network that resembles Petri nets. Hierarchy is created by refining dataflow blocks, not states as in DFCharts. Hence preemption of dataflow networks is not possible in Funstate. The HCFSM part drives the dataflow part which is not the case in DFCharts where SDF graphs work independently. Unlike DFCharts, Funstate is not intended for the specification level, its purpose is to represent designs internally.

Composite signal flow [101] also uses two different models to handle control-dominated and data-dominated parts, but in contrast to Funstate and DFCharts, those two models cannot be nested, they can only be connected at the top level.

Ptolemy [20] is an environment where several models of computation are hierarchically combined. At each hierarchical level blocks have to obey the semantics of a single model of computation, but internally each block can be refined into a system that behaves according to some other model. The closest subset of Ptolemy to DFCharts is *charts [102] where the focus is on mixing FSMs with other models. With purely hierarchical heterogeneity it may be difficult to devise a meaningful communication mechanism between the outer and inner models. The inner model may lose some properties while adjusting to the outer model. This is especially the case when the two models perform vastly different communication on their ports. For example, when a
communicating sequential processes (CSP) block is refined by a synchronous reactive (SR) block, the interface must translate non-blocking read and non-blocking write of SR into blocking read and blocking write of CSP.

These issues become apparent when the frequency relay is specified in Ptolemy. Figure 3.12 shows a possible solution. The top level MoC is FSM. State S2 is refined by a network of blocks that follow the semantics of the synchronous reactive model. All blocks are internally FSMs except for find peaks, which is refined into an SDFG. While synchronous communication is a perfect match for the FSMs, it is not suitable for the
SDFG. Since it also has to conform to the synchrony hypothesis, its iteration is assumed to be instantaneous and must be executed simultaneously with other SR blocks. As a result, an inefficient implementation is likely to emerge from this specification. Another option is to use a dataflow model at the second hierarchical level such as KPN. This would be favourable for the SDFG. The interface between KPN and SDF would be straightforward since both models use FIFO channels for communication. However, when blocking reads of KPN are imposed on parameter settings FSM, it cannot react to external events. With parallel heterogeneity used in DFCharts, FSMs are free to react to external events and SDF graphs run at their own speed.

A third option could be the refinement of S2 by the discrete event (DE) [9] model. DE is a highly expressive model that can accommodate a variety of communication mechanisms. The problems described above would be reduced. However, DE can hardly be used for synthesis [5],[102] because of the extensive sorting of time stamps that would be very expensive to implement. Thus, we would end up with a specification that can only be simulated.

Communicating reactive state machines (CRSM) [103] also extend Argos by an asynchronous parallel operator which connects parallel FSMs by rendezvous channels. The purpose of the asynchronous parallel operator in CRSM is to connect parts in a distributed system, while in DFCharts it is used to connect control-dominated and data-dominated parts that are physically close. Another important difference is that in CRSM the asynchronous parallel operator can be applied only at the top level (in a locally synchronous but globally asynchronous (GALS) manner) while in DFCharts it can be applied at any hierarchical level.

Mode Automata [104] extends the scope of Argos towards signal processing systems by combining it with another synchronous language Lustre [16]. While Lustre has a dataflow flavour due to its declarative statements, its semantics is quite different than that of SDF and other dataflow models derived from Kahn process networks. As in other synchronous languages, the execution of Lustre consists of a series of ticks where all computations and communications are assumed instantaneous. Thus, there is no need for buffers as in SDF. Unlike Mode Automata, DFCharts is not completely synchronous.
since it contains parts that operate at different speeds. Consequently, it should have wider implementation space. This means at the same time that verification of DFCharts models is more extensive.
Chapter 4

Semantics of DFCharts

In this chapter we present the formal semantics of DFCharts. Section 4.1 discusses the automata semantics, introduced first in [105], where the behaviour of a complete specification is expressed as an FSM. This section is divided into seven sub-sections. Section 4.1.1 introduces multiclock FSMs that are used as inputs to DFCharts operators. Sections 4.1.2 to 4.1.5 define the semantics of four DFCharts operators: synchronous parallel, asynchronous parallel, localization\hiding and refinement. DFCharts operators are defined in a similar style as Argos [24] operators. However, because of the use of multiclock FSMs, their operation is quite different. Section 4.1.6 defines with a simple language how the operators may be applied on multiclock FSMs. As seen in Chapter 3, an SDFG and an FSM are connected with the asynchronous parallel operator. According to the semantics of the asynchronous parallel operator in section 4.1.3, it operates only on FSMs like all other operators. For this reason, an SDFG is represented as an FSM so that it can be combined with “real FSMs”. This is the topic of section 4.1.7. Since an SDFG proceeds at its own speeds, an FSM that represents an SDFG (“SDF FSM”) is triggered by a clock that is different from the clock of real FSMs. Thus, when a “real FSM” and “SDF FSM” are combined, a multiclock FSM results. In the definitions of operators, which comprise DFCharts automata semantics, rendezvous is treated simply as an event that triggers a transition. What exactly happens on the channel is irrelevant. This is the topic of section 4.2. It examines in detail the ordering of events on a channel within the Tagged Signal Model (TSM) framework. The analysis
leads to understanding of how an array variable produces multiple SDF tokens and vice versa. Data transfer from SDF to FSM is the topic of section 4.2.1, while section 4.2.2 deals with data transfer from FSM to SDF. The TSM semantics of DFCharts was previously described in [106]. Finally, Section 4.3 examines the effect of clock speeds on DFCharts behaviour.

4.1 Automata semantics

In DFCharts automata semantics, the behaviour of a specification is determined by its equivalent, flat FSM, as in Argos. The equivalent FSM is obtained by combining FSMs and SDFGs using four operators: synchronous parallel (||), asynchronous parallel (\), hiding (\) and refinement (\) marks the state refinement by FSMs with or without SDFGs while \) marks the state refinement only by an SDFG). All four DFCharts operators are associative. The synchronous parallel operator is also commutative. The equivalent FSM is constructed bottom-up, by starting from the lowest hierarchical levels and moving upwards to the top level. Inside a state all FSMs are firstly combined by the synchronous parallel operator with the hiding operator applied if there are any local signals. Then the equivalent FSM is combined with SDFGs by the asynchronous parallel operator to produce another single flat FSM. The resulting FSM is then combined with the higher-level FSM by the refinement operator.

Of course, the process described above would not be possible without representing the operation of an SDFG as an FSM, which will be discussed in section 4.1.7. While “real FSMs” are all driven by the same clock, which we will call gclk (global clock), “SDF FSMs” are driven by their own individual clocks. Each clock in DFCharts semantics is a sequence of ticks. As in synchronous languages, each tick denotes a reaction. Clocks of “SDF FSMs” synchronize with gclk only when a rendezvous occurs. It is important to mention that gclk can start and stop SDF clocks. The relation between gclk and SDF clocks is illustrated with the example in Figure 4.1. Ticks occur simultaneously only at rendezvous points when two clocks are synchronized, as indicated by dotted lines. Ticks cannot occur simultaneously by chance. This assumption needs to be made to ensure deterministic behaviour. As indicated in Figure 4.1, it is possible that ticks of two SDF
clocks occur simultaneously, but this is only the result of their simultaneous synchronization with \( gclk \). SDF clocks cannot synchronize directly. It should be noted that SDF is not the only model that can be abstractly represented as an FSM and connected with FSMs through channels. Other dataflow models which have clearly defined iteration can easily be incorporated.

The steps for building the equivalent FSM \( E \) for the example in Figure 3.1 which is reproduced here are shown in Figure 4.2. \( E1 \) to \( E7 \) are intermediate equivalent FSMs created in the process. Each step indicates the operator that is applied. Also the clock(s) for each FSM are given in brackets. FSMs that have more than one clock are called *multiclock FSMs*. FSMs that refine the same state are placed in a rectangle. We explain a few steps in the beginning. At the lowest level of hierarchy are FSM6, FSM7 and SDF2. FSM6 and FSM7 are firstly combined with the synchronous parallel operator producing the equivalent FSM \( E1 \). \( E1 \) and SDF2-FSM are then combined with the asynchronous parallel operator producing the equivalent FSM \( E2 \). Since the clock of \( E1 \) is \( gclk \) and the clock of SDF2-FSM is \( sdfclk_2 \), \( E2 \) must have both \( gclk \) and \( sdfclk_2 \). In the next step the hierarchical operator is applied on FSM4 and \( E2 \) to produce \( E3 \). The process continues until \( E \) is reached, which represents the behaviour of the whole specification in Figure 3.1. It is important to observe that only the asynchronous parallel operator can produce a multi-clock FSM from two single-clock FSMs. The
synchronous parallel and refinement operators produce a multiclock FSM if at least one of their input FSMs is multiclock.

Figure 3.1: A DFCharts Specification
Figure 4.2: Building the equivalent FSM for the example in Figure 3.1
We divide FSMs that appear in DFCharts automata semantics in two groups:

- **gclk FSMs**: These can be a single clock FSMs driven by gclk or multiclock FSMs that have other clocks besides gclk.

- **non-gclk FSMs**: These are single clock FSMs used to model SDF and possibly other dataflow models.

In Figure 4.2 SDF2-FSM and SDF1-FSM are non-gclk FSMs. All other FSMs are gclk FSMs. A non-gclk FSM may also be derived from multiple FSMs. Figure 4.3 shows how SDF2-FSM from Figure 4.2 may be constructed. A more detailed and precise representation of an SDFG will be discussed in section 4.1.7. At this point it is only important to realize that a non-gclk FSM can be composed from other non-gclk FSMs that are connected by synchronous parallel and refinement operators. The hiding operator is also applied if local signals exist.

![Diagram](attachment://diagram.png)

**Figure 4.3**: Possible composition of a non-gclk FSM

Since multiclock FSM is too large to be drawn on a single page we present a simple multiclock FSM in Figure 4.4. Due to space constraints gclk is written as $k$. The additional clock in the FSM is $k_1$. This notation will be followed in other figures showing multiclock FSMs – gclk will be labelled as $k$ while other clocks will be labelled $k_1, k_2...$ At the beginning of each transition label, it is indicated which clock drives it. Synchronization of clocks is indicated by using brackets in the clock label. All clocks
that are synchronized with $k$ are placed in brackets. In a multicycle FSM, inputs are tied to specific clocks. At each tick of $k$, input $a$ is read. On the other hand, input $b$ is tied to clock $k1$. When $k$ and $k1$ are synchronized, in which case the clock label is $<k(k1)>$, $a$ and $b$ are read at the same time. As in section 2.4, a dot between inputs in monomial means logical AND, while a bar over an input means logical NOT.

![Figure 4.4: A multicycle FSM](image)

An alternative to having multicycle FSMs is to treat clocks as boolean inputs. A base clock is introduced which is faster than any other clock. When a tick of the base clock occurs a transition will be taken if both its clock and input monomial are true. Since there is effectively a single clock triggering all transitions the semantics is likely to be simpler. This approach is used in Multicycle Esterel [58].

While it is desirable for the semantics to be as simple as possible, it should also reflect the true behaviour of the system as closely as possible. The base clock is fictitious. It does not show up in implementation. Moreover, it may be difficult to think of clocks as
inputs. In pure hardware implementation which Multiclock Esterel targets, clocks do appear as input signal lines in the synthesized circuit. This is not the case in software implementation where a clock tick could represent the execution of several instructions. In DFCharts, treating clocks as inputs is further complicated by the synchronization of clocks. When two FSMs are ready to communicate from rendezvous states, the next ticks of their clocks must occur simultaneously. If the clocks are viewed as inputs, they must be both true in the next tick of the fictitious base clock. Forcing clocks to have the same status means that they must be considered as a special kind of inputs. As a result, the definitions of the operators for composition would have to contain conditions attached to clocks. In the end, the single clock semantics may not be significantly simpler than the semantics based on multiclock FSMs that is presented in the following sections.

Reading asynchronous clocks like any other external inputs could result in many unnecessary transitions that could make analysis more difficult. As seen in Figure 4.4, there are four transitions going out of every state where clocks k and k1 are not synchronized: \( <k > a, <k > \neg a, <k1 > b, <k1 > \neg b \). If k and k1 are treated as inputs, the total number of inputs becomes four resulting in sixteen transitions:

\[
\begin{align*}
  &k.k1.a.b,k.k1.a.\neg b,k.k1.a.b,k.k1.a.\neg b, \\
  &k.k1.a.b,k.k1.a.\neg b,k.k1.a.b,k.k1.a.\neg b
\end{align*}
\]

Including clocks in transition monomials is much more efficient in a multiclock synchronous system where all clocks are derived from the fastest clock. Figure 4.5 shows an example of multiclock synchronous system.

\[
\begin{array}{|c|c|c|c|c|c|}
  \hline
  clk3 & & & & & \\
  clk2 & & & & & \\
  clk1 & & & & & \\
  \hline
\end{array}
\]

Figure 4.5: A multiclock synchronous system
4.1.1 FSM with variables

**Definition 4.1.1**: Finite state machine with variables

\[ A = (CLK, Q, q_0, I, R, O, V, C, T, RQ, Proc) \]

\(CLK\) is the set of clocks that drive FSM transitions. For gclk FSMs, \( gclk \in CLK \). For non-gclk FSMs, \( gclk \notin CLK \) and \( |CLK| = 1 \). \( Q \) is set of states where \( q_0 \) is the initial state. \( I \) is the set of input signals. Each input signal can either be present (true) or absent (false). \( R \) is the set of channel status signals (CS signals). It is composed of external CS signals \( R_E \) and internal CS signals \( R_I \), \( R = R_E \cup R_I \). In DFCharts automata semantics, CS signals are tested in the same way as input signals. When a clock tick occurs, a CS signal is true if a rendezvous is happening on the corresponding channel and false otherwise. \( O \) is the set of output signals. Each output signal can either be emitted (true) or not emitted (false). \( V \) is the set of variables. \( C \) is the set of conditions on variables. Like input signals, CS signals and output signals, conditions are Boolean; they can be true or false. \( T \) is the set of transitions. \( RQ \) is the rendezvous mapping function. \( T \) and \( RQ \) will be defined below. \( Proc \) is the set of procedures on variables.

\( I, O, V, C, Proc \) and \( R_E \) are partitioned (divided into disjoint subsets) by clocks. On the other hand, internal CS signals \( R_I \) are shared among clocks as they are used for synchronization. Internal CS signals disappear when the asynchronous parallel operator is applied (section 4.1.3).

It was mentioned in section 3.1.1 that each rendezvous state is used for communication on a single channel. However, when an equivalent FSM is obtained by combining two or more FSMs, a state may be linked to multiple channels. \( RQ \), defined below, is a function that maps CS signals to states. Each state is mapped to zero or more CS signals.

**Definition 4.1.2**: Rendezvous mapping function

\[ RQ : q \rightarrow 2^e \]
Definition 4.1.3: FSM transitions

\[ T \subseteq CLK \times 2^{CLK} \times Q \times B(I') \times B(C') \times B(R') \times 2^O \times Proc \times Q \]

A transition is driven by a clock \( clk \) taken from the set \( CLK \). In addition, in the case of \( gc\k \) multiclock FSMs, there could be a set of clocks taken from \( 2^{CLK} \) (power set of \( CLK \)) that have to synchronize with \( clk \) when the transition occurs. This is applicable only when \( clk = gc\k \). \( B(I') \) where \( I' \subseteq I \) is the set of boolean formulas on input signals that are bound to \( clk \). Each formula is a complete monomial, or in other words a conjunction that has to contain every input signal in \( I' \). \( B(C') \) where \( C' \subseteq C \) is the set of Boolean formulas on variable conditions that are bound to \( clk \). Each formula is a complete monomial. \( B(R') \) where \( R' \subseteq R \) is the set Boolean formulas (again complete monomials) on CS signals that are linked to the source state of the transition and bound to \( clk \).

We can write a transition as a tuple \((clk, q, i, o, p, q')\). \( i = m.c.r \) where \( m, c \) and \( r \) are monomials over input signals, conditions on variables and CS signals respectively. The dot between monomials denotes the AND operation exactly in the same way as inside monomials. \( o \) and \( p \) denote output signals and procedures.

Complete monomials are required for the analysis of determinism and reactivity. DFCharts examples given in section 3.1, never contained complete monomials. The requirement to specify complete monomials would significantly increase specification time especially when there are many conditions on variables. Fortunately, it is not necessary since complete monomials can always be produced from incomplete monomials by adding transitions. We illustrate this with two examples in Figures 4.6 and 4.7. The first specification (Figure 4.6 (a)) has one input signal and two conditions on variables, the second (Figure 4.7 (a)) has two input signals and one channel. In both cases, all transitions are driven by \( gc\k \). In Figure 4.6 (b), \( b \) and \( c \) stand for conditions \([v>4]\) and \([v<6]\), while \( p \) stands for procedure \( \{v=v+1\} \). In Figure 4.7 (b), CS signal \( c \) corresponds to \( ch1\).
It can be observed in Figure 4.6(b) that data abstraction is used when conditions on variables are represented since values of variables are not visible. For each condition, it only matters whether it is true or false. Explicit representation of variables would make any analysis including causality related properties of determinism and reactivity impossible since each variable can take an infinite number of values. A similar approach for data abstraction is used in Esterel Studio [18]. There are many other approaches for ensuring that the state space is finite including symbolic bisimulation [107] in CCS and region graphs in timed automata [108]. In each state all input signals and all conditions are checked. However, this is not the case with CS signals. A CS signal is only checked in the states it is associated with. The variability in the number of outgoing transitions is seen Figure 4.7(b) where S1 has 8 outgoing transitions while S2 has 4 outgoing transitions. For definitions of reactivity and determinism, the variability in inputs from
state to state does not pose a problem. However, multiple clock domains are more difficult to deal with.

An obvious approach in handling reactivity and determinism for a multi-clock FSM is to check them separately for each clock domain. Then we can state that an FSM is deterministic and reactive if determinism and reactivity is satisfied in each clock domain. In this discussion we are not concerned with the relation between clocks. We only assume that ticks of two different clocks never occur simultaneously unless the clocks are synchronized. In section 4.3, we examine how the relative speeds of clocks affect behaviour.

Let $MQC$ denote a function that returns the set of all complete monomials $H$ that are applicable to state $q$ and clock $clk$ i.e. $H = MQC(q, clk)$. $|H| = 2^n$ where $n$ is the number of inputs read by clock $clk$ in state $q$. Determinism and reactivity are defined as follows.

**Definition 4.2**: An FSM is deterministic if and only if

$$\forall q \in Q, \forall clk \in CLK, \forall i \in MQC(q, clk), (clk, q, i, o_1, p_1, q'_1) \in T \land (clk, q, i, o_2, p_2, q'_2) \in T \Rightarrow (o_1 = o_2) \land (p_1 = p_2) \land (q'_1 = q'_2)$$

In every state, for every clock domain, no two transitions may have the same input combination in order for the FSM to be deterministic.

**Definition 4.3**: An FSM is reactive if and only if

$$\forall q \in Q, \forall clk \in CLK, \forall i \in MQC(q, clk), \exists (clk, q, i, o, p, q') \in T$$

In every state, for every clock domain, there has to be at least one transition for each input combination in order for the FSM to be reactive.

Before giving the definitions of the operators, we present synchronous product and asynchronous product (or interleaving), two well known methods used for composing
single clock FSMs. Synchronous parallel, asynchronous parallel and refinement operators use both synchronous product and interleaving.

Synchronous product is shown in Figure 4.8. The outgoing transitions from states A and B are synchronously combined to produce the outgoing transitions from state AB. This means that each transition monomial is state AB is obtained by ANDing monomials from states A and B. In Figure 4.8 all transitions are driven by the same clock, but synchronous product may also be applied when two different clocks are synchronized as shown in Figure 4.9. Asynchronous product is simply interleaving of transitions driven by two different clocks when they are not synchronized. It is shown in Figure 4.10. The transitions from states A and B are unchanged in state AB, they are simply copied. It is important to emphasize that transitions driven by \( k \) and \( k_1 \) are never enabled at the same time. Thus, the asynchronous product is deterministic due to the explicit use of clocks.

![Figure 4.8: Synchronous product with a single clock](image)

![Figure 4.9: Synchronous product with two synchronized clocks](image)

![Figure 4.10: Asynchronous product](image)
When the asynchronous parallel and hiding operators are applied, internal signals disappear. The removal of signals is done with the hiding function defined below.

**Definition 4.4: Hiding Function**

\[ HF : M \times 2^S \rightarrow M \]

S is the set of all inputs. M is the set of all possible monomials that can be formed from inputs in set S. \( HF(m, I) = m' \) where \( m' \) is the monomial that is obtained when signals in set \( I \) are removed from monomial \( m \).

We also define additional notation for monomials before the definitions of the operators. For a monomial \( m \), \( m^a \) is the set of all signals that appear in \( m \), \( m^+ \) is the set of signals that appear as positive elements in \( m \) while \( m^- \) the set of signals that appear as negative elements in \( m \). For example, for \( m = a \bar{b} \bar{c} \), \( m^a = \{a, b, c\} \), \( m^+ = \{a\} \) and \( m^- = \{b, c\} \).

### 4.1.2 Synchronous parallel operator

The synchronous parallel operator can connect two gelk FSMs or two non-gelk FSMs driven by the same clock. FSMs connected by the synchronous parallel operator must not have any common CS signals. When the definition below is applied on two gelk FSMs, \( mclk \) stands for \( gelk \). When two non-gelk FSMs are involved, \( mclk \) represents the single clock of the two FSMs. The transition set of the FSM produced by the synchronous parallel operator consists of three types of transitions, grouped into subsets that are marked in the definition below. Each subset will be briefly described after the definition.

**Definition 4.5: Synchronous parallel operator**

\[
(CLK_1, Q_1, q_{01}, I_1, R_1, O_1, V_1, C_1, T_1, RQ_1, Proc_1) || \\
(CLK_2, Q_2, q_{02}, I_2, R_2, O_2, V_2, C_2, T_2, RQ_2, Proc_2)
\]
\[
T = (CLK_1 \cup CLK_2, Q_1 \times Q_2, (q_{01}, q_{02}), I_1 \cup I_2, R_1 \cup R_2, O_1 \cup O_2, V_1 \cup V_2, C_1 \cup C_2
\]

\[T, RQ, Proc_1 \cup Proc_2)\]

where

\[
RQ_1(q_1) = S_1 \land RQ_2(q_2) = S_2 \to RQ(q_1q_2) = S_1 \cup S_2
\]

and

\[
T = \{ (mclk, CLK'_1 \cup CLK'_2, (q_1, q_2), m_1 \land m_2, c_1 \land c_2, r_1 \land r_2, O'_1 \cup O'_2, Proc'_1 \cup Proc'_2
\]

\[
, (q'_1, q'_2)) \mid
\]

\[
(mclk, CLK'_1, q_1, m_1, c_1, r_1, O'_1, Proc'_1, q'_1) \in T_1,
\]

\[
(mclk, CLK'_2, q_2, m_2, c_2, r_2, O'_2, Proc'_2, q'_2) \in T_2, m_1 \land m_2 \neq false, c_1 \land c_2 \neq false\}^1
\]

\[
\cup \{ (clk_1, \emptyset, (q_1, q_2), m_1, c_1, r_1, O'_1, Proc'_1, (q'_1, q'_2)) \mid
\]

\[
(clk_1, \emptyset, q_1, m_1, c_1, r_1, O'_1, Proc'_1, q'_1) \in T_1, clk_1 \neq mclk\}^2
\]

\[
\cup \{ (clk_2, \emptyset, (q_1, q_2), m_2, c_2, r_2, O'_2, Proc'_2, (q'_1, q'_2)) \mid
\]

\[
(clk_2, \emptyset, q_2, m_2, c_2, r_2, O'_2, Proc'_2, q'_2) \in T_2, clk_2 \neq mclk\}^3
\]

1. Both FSMs take their \textit{mclk} driven transitions simultaneously.
2. The first FSM makes a transition driven by a clock different than \textit{mclk}, while the second one is idle.
3. The second FSM makes a transition driven by a clock different than \textit{mclk}, while the first one is idle.

The conditions \(m_1 \land m_2 \neq false\) and \(c_1 \land c_2 \neq false\) in the above definition are important since they remove unwanted transitions that arise when the two FSMs have common inputs. For a common input \(a\), four combinations are produced: \(a.a\), \(\overline{a}.a\), \(a.\overline{a}\), \(\overline{a}.\overline{a}\). \(a.\overline{a}\) and \(a.\overline{a}\) must be removed since they have no meaning. They always evaluate to false. They would also be a source of non-determinism since they represent the same monomial. After the removal, the number of transitions is halved. For \(n\) common inputs, the number of transitions is reduced by \(2^n\). While only synchronous
product appears in the parallel operator of Argos, the synchronous parallel operator of DFCharts involves both synchronous product (transition subset 1) and interleaving (transitions subsets 2 and 3). Subsets 2 and 3 are empty unless the asynchronous parallel operator has been applied beforehand at a lower hierarchical level.

Figures 4.11, 4.12 and 4.13 show FSMs A1, A2 and A3 that are used to illustrate the application of the synchronous parallel operator. A3 is produced when A1 and A2 are merged by the synchronous parallel operator. In the diagram showing A3, transitions that connect the same states are represented by a single line. Transition labels are defined below the diagram. According to definition 4.5 all transitions that are driven by $k$ (clock labels $<k>$, $<k(k1)>$, $<k(k2)>$ and $<k(k1,k2)>$) are in subset 1; all transitions that are driven by $k_1$ (clock label $<k1>$) are in subset 2; all transitions that are driven by $k_2$ (clock label $<k2>$) are in subset 3.

Figure 4.11: FSM A1

Figure 4.12: FSM A2
Transitions from S11S21:
\[ T1 = \{ <k2 > \bar{a}; <k2 > d / o4; <k(k1) > a.\bar{b}.c / o1; <k(k1) > a.b.c / o1 \} \]
\[ T2 = \{ <k(k1) > a.\bar{b}.\bar{c} / o1; <k(k1) > a.b.\bar{c} / o1 \} \]
\[ T3 = \{ <k(k1) > \bar{a}.\bar{b}.c; <k(k1) > \bar{a}.b.c \} \]
\[ T4 = \{ <k(k1) > \bar{a}.\bar{b}.\bar{c}; <k(k1) > \bar{a}.b.\bar{c} \} \]

Transitions from S12S21:
\[ T5 = \{ <k1 > \bar{b}; <k1 > b / o2; <k2 > \bar{a}; <k2 > d / o4; <k > \bar{a}.c \} \]
\[ T6 = \{ <k > a.c \} \]
\[ T7 = \{ <k > \bar{a}.\bar{c} \} \]
\[ T8 = \{ <k > a.\bar{c} \} \]

Transitions from S12S22:
\[ T9 = \{ <k1 > \bar{b}; <k1 > b / o2 \} \]
\[ T10 = \{ <k(k2) > \bar{a}.\bar{c}.\bar{d}; <k(k2) > \bar{a}.\bar{c}.d; <k(k2) > \bar{a}.c.d / o3; <k(k2) > \bar{a}.c.d / o3 \} \]
\[ T11 = \{ <k(k2) > a.\bar{c}.\bar{d}; <k(k2) > a.\bar{c}.d; <k(k2) > a.\bar{c}.d / o3; <k(k2) > a.\bar{c}.d / o3 \} \]

Transitions from S11S22:
\[ T12 = \{ <k(k1,k2) > a.\bar{b}.\bar{c}.\bar{d} / o1; <k(k1,k2) > a.\bar{b}.\bar{c}.d / o1; <k(k1,k2) > a.\bar{b}.c.d / o1, o3; <k(k1,k2) > a.\bar{b}.c.d / o1, o3; <k(k1,k2) > a.\bar{b}.\bar{c}.d / o1, o3; <k(k1,k2) > a.\bar{b}.\bar{c}.d / o1, o3; <k(k1,k2) > a.b.\bar{c}.d / o1, o3; <k(k1,k2) > a.b.\bar{c}.d / o1, o3; <k(k1,k2) > a.b.\bar{c}.d / o1, o3; <k(k1,k2) > a.b.\bar{c}.d / o1, o3 \} \]
\[ T13 = \{ <k(k1,k2) > \bar{a}.\bar{b}.\bar{c}.\bar{d}; <k(k1,k2) > \bar{a}.\bar{b}.\bar{c}.d; <k(k1,k2) > \bar{a}.\bar{b}.c.d / o3; <k(k1,k2) > \bar{a}.\bar{b}.c.d / o3; <k(k1,k2) > \bar{a}.b.\bar{c}.\bar{d}; <k(k1,k2) > \bar{a}.b.\bar{c}.\bar{d}; <k(k1,k2) > \bar{a}.b.\bar{c}.d; <k(k1,k2) > \bar{a}.b.\bar{c}.d \} \]

Figure 4.13: FSM A3=A1||A2
Theorem 4.1

When the synchronous parallel operator is applied on two deterministic and reactive FSMs the resulting FSM is deterministic and reactive.

Proof

$mclk$:

Let $I_{q_1}, C_{q_1}, R_{q_1}$ be the sets of input signals, variable conditions and CS signals bounded to $mclk$ respectively for state $q_1$ and similarly $I_{q_2}, C_{q_2}, R_{q_2}$ for state $q_2$. Let $ci$ be the number of common inputs (signals and conditions), $ci = |I_{q_1} \cap I_{q_2}| + |C_{q_1} \cap C_{q_2}|$. The input signals, variable conditions and CS signals for the equivalent state $q_1 q_2$ are $I_{q_1 q_2} = I_{q_1} \cup I_{q_2}$, $C_{q_1 q_2} = C_{q_1} \cup C_{q_2}$, and $R_{q_1 q_2} = R_{q_1} \cup R_{q_2}$. The total number of inputs in state $q_1 q_2$ is the number of outgoing transitions is $nt = 2^{ni}$.

From the definition of subset 1 of $T$, it is evident that the number of outgoing transitions from state $q_1 q_2$ is equal to $nt = (nt_1 \cdot nt_2) / 2^{ci}$ where $nt_1$ and $nt_2$ are the numbers of outgoing transitions from the states $q_1$ and $q_2$ respectively. Since both input FSMs are deterministic and reactive $n_1 = 2^{(|I_{q_1}| + |C_{q_1}| + |R_{q_1}|)}$ and $n_2 = 2^{(|I_{q_2}| + |C_{q_2}| + |R_{q_2}|)}$. Hence, $n = (2^{(|I_{q_1}| + |C_{q_1}| + |R_{q_1}|)} \cdot 2^{(|I_{q_2}| + |C_{q_2}| + |R_{q_2}|)}) / 2^{ci} = 2^{(|I_{q_1}| + |C_{q_1}| + |C_{q_2}| + |R_{q_1}| + |R_{q_2}| - ci)} = 2^{ni}$.

For non-determinism to be present, there would have to be two transitions out of $q_1 q_2$ with the same monomials, $m_1 c_1 r_1 \wedge m_2 c_2 r_2 = m'_1 c'_1 r'_1 \wedge m'_2 c'_2 r'_2$, which requires $m_1 c_1 r_1 = m'_1 c'_1 r'_1$ and $m_2 c_2 r_2 = m'_2 c'_2 r'_2$. However, this cannot be true since the assumption is that both input FSMs are deterministic. If each transition in $T$ is unique and there are $2^{ni}$ transitions then all input combinations are defined which means that reactivity is also satisfied.
Other clocks:
Transitions that appear in subsets 2 and 3 are simply copied as in Figure 4.10. Thus
determinism and reactivity are preserved.

4.1.3 Asynchronous parallel operator

In the definition below \( CR \) is the set of common CS signals between the two FSMs,
\( CR = R_1 \cap R_2 \). \( CR \) must not be empty. The left operand has to be a gclk FSM while the
right operand has to be a non-gclk FSM. The transition set \( T \) is divided into four subsets
that are explained below.

**Definition 4.6** Asynchronous parallel operator

\[
(\text{CLK}_1, Q_1, q_{01}, I_1, R_1, O_1, V_1, C_1, T_1, RQ_1, \text{Proc}_1) \parallel \\
(\text{CLK}_2, Q_2, q_{02}, I_2, R_2, O_2, V_2, C_2, T_2, RQ_2, \text{Proc}_2) \\
= (\text{CLK}_1 \cup \text{CLK}_2, Q_1 \times Q_2, (q_{01}, q_{02}), I_1 \cup I_2, (R_1 \cup R_2) \setminus CR, O_1 \cup O_2, V_1 \cup V_2, C_1 \cup C_2 \\
T, RQ, \text{Proc}_1 \cup \text{Proc}_2)
\]

where

\[
RQ(q_1) = S_1 \land RQ(q_2) = S_2 \rightarrow RQ(q_1, q_2) = (S_1 \cup S_2) \setminus (S_1 \cap S_2)
\]

\[
T = \{ (\text{gclk}, \text{CLK}', (q_1, q_2), m_1, c_1, HF(r_1, r_1^{c} \cap CR), O'_1, \text{Proc}'_1, (q'_1, q'_2)) | \\
(\text{gclk}, \text{CLK}', q_1, m_1, c_1, r_1, O'_1, \text{Proc}'_1, q'_1) \in T_1, \\
RQ(q_1) \cap RQ(q_2) = \emptyset, r_1^{c} \cap CR = \emptyset \}^{1}
\]

\[
\cup \{ (\text{gclk}, \text{CLK}', \cup \text{clk}_2, (q_1, q_2), m_1 \land m_2, c_1 \land c_2, HF(r_1 \land r_2, (r_1 \land r_2)^{c} \cap CR), \\
O'_1 \cup O'_2, \text{Proc}'_1 \cup \text{Proc}'_2, (q'_1, q'_2)) | \\
(\text{gclk}, \text{CLK}', q_1, m_1, c_1, r_1, O'_1, \text{Proc}'_1, q'_1) \in T_1, \\
(\text{clk}_2, \emptyset, q_2, m_2, c_2, r_2, O'_2, \text{Proc}'_2, q'_2) \in T_2, \\
RQ(q_1) \cap RQ(q_2) \neq \emptyset, (r_1 \land r_2)^{c} \cap CR = RQ(q_1) \cap RQ(q_2), r_1 \land r_2 \neq \text{false} \}^{2}
\]

74
\( \cup \{ (clk_1, \emptyset, (q_1, q_2), m_1, c_1, r_1, O_1', Proc_1', (q_1', q_2')) \mid (clk_1, \emptyset, q_1, m_1, c_1, r_1, O_1', Proc_1', q_1') \in T_1 \}^3 \)

\( \cup \{ (clk_2, \emptyset, (q_1, q_2), m_2, c_2, r_2, O_2', Proc_2', (q_1', q_2')) \mid (clk_2, \emptyset, q_2, m_2, c_2, r_2, O_2', Proc_2', q_2') \in T_2, RQ(q_1) \cap RQ(q_2) = \emptyset, r_2^+ \cap CR = \emptyset \}^4 \)

1. The first FSM makes a gclk transition while the second one is idle. In the current states \( q_1 \) and \( q_2 \) the two FSMs have no common CS signals as denoted by \( RQ(q_1) \cap RQ(q_2) = \emptyset \). If \( q_1 \) has a CS signal that belongs to the CR set, a rendezvous on the corresponding channel will never happen while the equivalent machine is in \( q_1 q_2 \). Therefore, all transitions where any common CS signal appears as a positive input are removed as denoted by the condition \( r_1^+ \cap CR = \emptyset \). Transitions where all common CS signals appear as negative inputs survive, but the common CS signals are removed with the hiding function as denoted by \( HF(r_1, r_1^a \cap CR) \).

2. The two FSMs are in the states which share at least one CS signal. \( ( RQ(q_1) \cap RQ(q_2) \neq \emptyset ) \). The next ticks of gclk and the clock of the second FSM will occur simultaneously. A transition survives if CS signals corresponding to common channels where rendezvous is about to happen (that belong to set \( RQ(q_1) \cap RQ(q_2) ) \) appear as positive inputs. This is denoted by the condition \( (r_1 \wedge r_2^+ \cap CR = RQ(q_1) \cap RQ(q_2) ) \). This condition implies at the same time that if a CS signal corresponds to a common channel where rendezvous will not happen in the next tick (belongs to set \( CR/(RQ(q_1) \cap RQ(q_2) ) \)) it has to appear as a negative input. All common CS signals are removed with the hiding function as denoted by \( HF(r_1 \wedge r_2^a \cap CR \neq RQ(q_1) \cap RQ(q_2) ) \). The condition \( r_1 \wedge r_2 \neq false \) removes all invalid monomials like \( \bar{a} \).

3. The first FSM makes a transition not driven by mclk. The second one is idle.

4. The second FSM makes a transition while the first one is idle. The two FSMs are in states which don’t share any CS signals. Therefore, only transitions where all common CS signals appear as negative inputs survive.
While the synchronous product applied in the synchronous parallel operator involves a single clock, the synchronous product applied in the asynchronous parallel operator involves two different clocks. Both operators use interleaving.

The asynchronous parallel operator in CRSM [103], which combines two single-clock FSMs, distinguishes three cases: the first FSM takes a transition while the second one is idle, the second FSM takes a transition while the first one is idle, both FSMs take transitions simultaneously. These three cases are also found in the asynchronous composition in CCS [14] and CSP [13]. In DFCharts, one of the input FSMs is multiclock in general. The transitions of the multiclock FSM that are driven by gc
\textsuperscript{cl}\textsubscript{k} have to be treated separately from those that are driven by other clocks, since only gc
\textsuperscript{cl}\textsubscript{k}-driven transitions are involved in rendezvous. Thus, there are four cases.

Figures 4.14, 4.15 and 4.16 show FSMs A4, A5 and A6 that are used to illustrate the application of the asynchronous parallel operator. In Figure 4.14 and 4.15, \( r \) is a CS signal. It is separated from input signals by “,” so that it is easy to distinguish. Comma means AND exactly like dot. A6 is produced when A4 and A5 are merged by the asynchronous parallel operator. According to definition 4.6, transitions in A6 can be classified as follows: all transitions that are driven by \( k \) where \( k \) is not synchronized with \( k_2 \) (clock labels \(<k>\) and \(<k(k_1)>)\) are in subset 1; all transitions that are driven by \( k \) where \( k \) is synchronized with \( k_2 \) (clock label \(<k(k_2)>)\) are in subset 2; all transitions that are driven by \( k_1 \) (clock label \(<k_1>)\) are in subset 3; all transitions that are driven by \( k_2 \) (clock label \(<k_2>)\) are in subset 4.

Figure 4.14: FSM A4
Figure 4.15: FSM A5

We look at how common CS signals are removed by the asynchronous parallel operator by considering two states of the equivalent FSM A6 in Figure 4.16, S11S21 where there is no rendezvous, and S11S22 where rendezvous occurs. There are four transitions driven by $k$ out of state S11 with monomials $\overline{a}.\overline{r}, a.\overline{r}, \overline{a}.r, a.r$. When S11 is combined with S21, transitions where $r$ appears as a positive input have to be removed, which leaves $\overline{a}.\overline{r}$ and $a.\overline{r}$, and after hiding $\overline{a}$ and $a$. The transitions that are driven by $k1$ and $k2$ are simply copied.

CS signal $r$ is common for states S11 and S22. Their outgoing transitions driven by $k$ and $k2$ are taken simultaneously because of rendezvous. There are sixteen such transitions initially but after conditions $(r_1 \wedge r_2)^+ \cap CR = RQ(q_1) \cap RQ(q_2)$ and $r_1 \wedge r_2 \neq false$ are applied only four survive as shown in Figure 4.16. The transitions that are driven by $k1$ are copied.

For each channel, it is known at compile-time in which state the rendezvous will happen. This is possible due to the absence of strong abort in DFCharts. For example, once FSM A6 enters state S11S22 the rendezvous on channel $r$ must happen in the next tick of $k$. If a weak abort occurs in the next tick due to a higher level transition, A6 will be allowed to react. Due to the way in which FSMs are composed in DFCharts, A6 can only be preempted by a transition that is driven by $k$. 
Transitions from S11S21:
T1 = {< k1 > b; < k1 > b / o1; < k > a; < k2 > c}
T2 = {< k2 > c}

Transitions from S12S21:
T3 = {< k(k1) > ab / o2; < k(k1) > ab / o2; < k(k1) > ab; < k(k1) > ab}
T4 = {< k2 > c}
T5 = {< k2 > c}

Transitions from S12S22:
T6 = {< k2 > c, < k2 > c}
T7 = {< k(k1) > ab / o2; < k(k1) > ab / o2; < k(k1) > ab; < k(k1) > ab}

Transitions from S11S22:
T8 = {< k(k2) > a; < k(k2) > a; < k(k2) > a; < k(k2) > a; < k(k2) > a}
T9 = {< k1 > b; < k1 > b / o1}

Figure 4.16: FSM A6

Theorem 4.2

When the asynchronous parallel operator is applied on two deterministic and reactive FSMs the resulting FSM is deterministic and reactive.
**Proof**

*mclk:*
When the equivalent FSM is in a state where rendezvous will occur in the next tick, mclk transitions are synchronously combined with the transitions of the other clock. It was already proved that synchronous product preserves determinism and reactivity. When a common CS signal is removed, other inputs are not affected. For a CS signal $r$ and monomial $m$, there are initially two possibilities: $\overline{r}.m$ and $r.m$. If a positive CS signal is removed, $\overline{r}.m$ remains and after hiding only $m$. If a negative CS signal is removed, $r.m$ remains and after hiding only $m$. In both cases, reactivity and determinism are preserved.

When the equivalent FSM is in a state where rendezvous cannot occur in the next tick, mclk driven transitions are copied. The removal of negative CS inputs does not affect determinism and reactivity as shown above.

*Other clocks:*
The same argument applies as in the second paragraph in the mclk part. Transitions are copied, while removal of negative CS signals does not affect determinism and reactivity.

4.1.4 Hiding operator

**Definition 4.7:** Hiding/localization operator

$$(CLK, Q, q_0, I, R, O, V, C, T, RQ, Proc) \setminus a = (CLK, Q, q_0, I \setminus a, R, O \setminus a, V, C, T', RQ, Proc)$$

where

$$T' = \{(clk, CLK', q, HF(m, \{a\}), c, r, O' \setminus a, Proc', q') \mid (clk, CLK', q, m, c, r, O', Proc', q') \in T, a \in O', a \in m\}$$

\[\]
\[ \circ \{(clk, CLK', q, HF(m, \{a\}), c, r, O', Proc', q')| (clk, CLK', q, m, c, r, O', Proc', q') \in T, a \notin O', a \in m^-\}^2 \]

1. If the local signal is present in the input part of the transition and emitted in the output part of the transition then the transition survives and the local signal is hidden
2. If the local signal is absent in the input part of the transition and not emitted in the output part of the transition then the transition survives and the local signal is hidden

Hiding was already seen in the context of the asynchronous parallel operator. That type of hiding should not be confused with the one defined above. The hiding operator from definition 4.7 is used to synchronize two FSMs that are connected by the synchronous parallel operator or refinement operator. The synchronization is performed with local signals. In contrast, the other type of hiding is built into the asynchronous parallel operator and works with CS signals. The presence of two types of hiding is not surprising as DFCharts employs two communication mechanisms, synchronous broadcast and rendezvous.

The hiding operator from the definition above preserves neither determinism nor reactivity. The causality cycles that can appear are exactly the same as those described in the Argos section.

4.1.5 Refinement operator

There are two variations of the refinement operator. Consequently, the definition below consists of two parts. The first type of the refinement operator, labelled \( \downarrow \), is used when both operands are either gcclk FSMs or non-gcclk FSMs. The second, labelled \( \uparrow \), is used when the left operand is a gcclk FSM while the right operand is a non-gcclk FSM. FSMs connected by the synchronous parallel operator must not have any common CS signals.
Definition 4.8 : Refinement operator

\[ (\text{CLK}_1, Q_1, q_{01}, I_1, R_1, O_1, V_1, C_1, T_1, RQ_1, \text{Proc}_1) \downarrow_q \]
\[ (\text{CLK}_2, Q_2, q_{02}, I_2, R_2, O_2, V_2, C_2, T_2, RQ_2, \text{Proc}_2) \]
\[ = (\text{CLK}_1 \cup \text{CLK}_2, (Q_1 \setminus \{q\}) \cup (q \times Q_2), q_{01}, I_1 \cup I_2, R_1 \cup R_2, O_1 \cup O_2, V_1 \cup V_2, C_1 \cup C_2 \]
\[ T, RQ, \text{Proc}_1 \cup \text{Proc}_2) \]

where

\[ RQ_1(q_1) = S_1 \land RQ_2(q_2) = S_2 \rightarrow RQ(q_1, q_2) = S_1 \cup S_2 \]

and

\[ T = \{(mclk, \text{CLK}_1', q_1, m_1, c_1, r_1, O'_1, \text{Proc}'_1, q_2) \in T_1 \mid q_1, q_2 \neq q\} \]
\[ \cup \{(mclk, \text{CLK}_1', q_1, m_1, c_1, r_1, O'_1, \text{Proc}'_1, (q, q_{02})) \mid (mclk, \text{CLK}_1', q_1, m_1, c_1, r_1, O'_1, \text{Proc}'_1, q) \in T_1\} \]
\[ \cup \{(mclk, \text{CLK}_2', (q, q_1), m_1 \land m_2, c_1 \land c_2, r_2, O'_2, \text{Proc}'_2, (q, q_2)) \mid (mclk, \emptyset, q, m_1, c_1, r_2, O'_2, \emptyset, q) \in T_1, m_1^+ = \emptyset, \]
\[ (mclk, \text{CLK}_2', q_1, m_2, c_2, r_2, O'_2, \text{Proc}'_2, q_2) \in T_2, m_1 \land m_2 \neq false, c_1 \land c_2 \neq false\} \]
\[ \cup \{(mclk, \text{CLK}_2', (q, q_1), m_1 \land m_2, c_1 \land c_2, r_2, O'_2 \cup O'_2, \text{Proc}'_2 \cup \text{Proc}'_1, (q, q_{02})) \mid (mclk, \emptyset, q, m_1, c_1, r_2, O'_2, \emptyset, q) \in T_1, m_1^+ = \emptyset, \]
\[ (mclk, \text{CLK}_2', q_1, m_2, c_2, r_2, O'_2, \text{Proc}'_2, q_2) \in T_2, m_1 \land m_2 \neq false, c_1 \land c_2 \neq false\} \]
\[ \cup \{(mclk, \text{CLK}_2', (q, q_1), m_1 \land m_2, c_1 \land c_2, r_2, O'_2 \cup O'_2, \text{Proc}'_2 \cup \text{Proc}'_1, q') \mid (mclk, \emptyset, q, m_1, c_1, r_2, O'_2, \emptyset, q') \in T_1, \]
\[ (mclk, \text{CLK}_2', q_1, m_2, c_2, r_2, O'_2, \text{Proc}'_2, q_2) \in T_2, m_1 \land m_2 \neq false, c_1 \land c_2 \neq false\} \]
\[ \cup \{(clk_1, \emptyset, q_1, m_1, c_1, r_1, O'_1, \text{Proc}'_1, q_2) \in T_1 \mid q_1, q_2 \neq q, clk_1 \neq mclk\} \]
\[ \cup \{(clk_2, \emptyset, (q, q_1), m_2, c_2, r_2, O'_2, \text{Proc}'_2, (q, q_2)) \mid (clk_2, \emptyset, q_1, m_2, c_2, r_2, O'_2, \text{Proc}'_2, q_2) \in T_2\} \]
As in the definition of the synchronous parallel operator, *mclk* stands for *gclk* when $\downarrow$ is applied on two *gclk* FSMS. In the case of non-*gclk* FSMS, *mclk* represents the single clock of the two FSMS. The transition set, which consists of seven subsets, is explained below.

1. *mclk* driven transitions of the first FSM that don’t involve the refined state.
2. *mclk* driven transitions of the first FSM that enter the refined state.
3. The second (refining) FSM makes an *mclk*-driven transition while the first FSM makes a *mclk* transition that loops back to the refined state. However the second FSM is not reset to its initial state $q_{02}$ since the transition of the first FSM does not involve any inputs that must be present ($m_i^+ = \emptyset$). Transitions where all inputs are absent are not specified by the programmer, they are inserted in order to obtain complete monomials. So they should not reset the refining FSM.
4. Same as 3, but this time at least one of the inputs in the first FSM’s transition is present so the refining FSM is reset to its initial state.
5. The refined state is left. Since the abort is weak the outputs of the second FSM are produced.
6. The first FSM makes the transition that is not driven by *mclk*. The refined state is not involved since it is always entered and exited by *mclk*-driven transitions.
7. The second FSM makes the transition that is not driven by *mclk* so the first FSM is idle.

The second part of the definition is used when the state of a *gclk* FSM is refined by a non-*gclk* FSM.

$$(CLK_1,Q_1,q_{01},I_1,R_1,O_1,V_1,C_1,T_1,RQ_1,Proc_1) \downarrow_q$$

$$(CLK_2,Q_2,q_{02},I_2,R_2,O_2,V_2,C_2,T_2,RQ_2,Proc_2)$$

$$= (CLK_1 \cup CLK_2, (Q_1 \setminus \{q\}) \cup (q \times Q_2), q_{01}, I_1 \cup I_2, R_1 \cup R_2, O_1 \cup O_2, V_1 \cup V_2, C_1 \cup C_2$$

$$T, RQ, Proc_1 \cup Proc_2$$

where
\[ RQ_1(q_1) = S_1 \land RQ_2(q_2) = S_2 \rightarrow RQ(q_1,q_2) = S_1 \cup S_2 \]

and

\[ T = \{(gclk,CLK',q_1,m_1,c_1,r_1,O_1',Proc'_1,q) \in T_1 | q_1,q_2 \neq q\} \]
\[ \cup \{(gclk,CLK',q_1,m_1,c_1,r_1,O_1',Proc'_1,(q,q_02)) \mid (gclk,CLK',q_1,m_1,c_1,r_1,O_1',Proc'_1,q) \in T_1\} \]
\[ \cup \{(gclk,\emptyset,(q,q_1),m_1,c_1,-,\emptyset,\emptyset,\emptyset,(q,q_1)) \mid (gclk,\emptyset,q,m_1,c_1,-,\emptyset,\emptyset,q) \in T_1,m_1^+ = \emptyset\} \]
\[ \cup \{(gclk,\emptyset,(q,q_1),m_1,c_1,-,O_1',Proc'_1,(q,q_02)) \mid (gclk,\emptyset,q,m_1,c_1,-,O_1',Proc'_1,q) \in T_1,m_1^+ \neq \emptyset\} \]
\[ \cup \{(gclk,\emptyset,(q,q_1),m_1,c_1,-,O_1',Proc'_1,q) \mid (gclk,\emptyset,q,m_1,c_1,-,O_1',Proc'_1,q) \in T_1\} \]
\[ \cup \{(clk_1,\emptyset,q_1,m_1,c_1,r_1,O_1',Proc'_1,q_2) \in T_1 | q_1,q_2 \neq q,clk_1 \neq mclk\} \]
\[ \cup \{(clk_2,\emptyset,q_1,m_2,c_2,r_2,O_2',Proc'_2,(q,q_2)) \mid (clk_2,\emptyset,q_1,m_2,c_2,r_2,O_2',Proc'_2,q_2) \in T_2\} \]

The transition set, which consists of seven subsets, is explained below.

1. \textit{gclk} driven transitions of the first FSM that don’t involve the refined state.
2. Transitions of the first FSM that enter the refined state.
3. The first FSM makes a transition that loops back to the refined state. The second FSM is not reset to its initial state \(q_{02}\) since the transition of the first FSM does not involve any input signals that must be present (\(m_i^+ = \emptyset\)).
4. Same as 3. but this time at least one of the input signals in the first FSM’s transition is present so the refining FSM is reset to its initial state.
5. The refined state is left. The abort is weak but there are no outputs from the second FSM since it does not react in this tick.
6. The first FSM makes a transition that is not driven by \textit{gclk}. The refined state is not involved since it is always entered and exited by \textit{mclk}-driven transitions.
7. The second FSM makes a transition while the first FSM is idle.

The definitions of the two versions of the hierarchical operator ($\downarrow$ and $\uparrow$) are similar but there is an important difference that creates the need for having two versions instead of just one. In the definition of $\downarrow$, synchronous product is present in subsets 3, 4 and 5 of T. In the definitions of $\uparrow$, synchronous product is completely absent. Transitions of the two input FSMs are just interleaved.

Figures 4.17, 4.18 and 4.19 show FSMs A7, A8 and A9 that are used to illustrate the application of the second version of the hierarchical operator. A9 is produced when A8 refines state S12 of A7.

![Figure 4.17: FSM A7](image)

![Figure 4.18: FSM A8](image)

![Figure 4.19: FSM A9=A7 $\uparrow_{S12}$ A8](image)
The first version of the hierarchical operator is illustrated with A9, A10 shown in Figure 4.20, and A11 shown in Figure 4.21. A11 is produced when state S11 of A9 is refined by A10.

Figure 4.20: FSM A10

Figure 4.21: FSM A11=A9↓s11 A10

Transitions from S11S31:
\[ T1 = \{ <k2> d/o4; <k2> \bar{d}; <k> \bar{a} \bar{c} \} \]
\[ T2 = \{ <k> \bar{a} c/o3 \} \]
\[ T3 = \{ <k> a \bar{c} / o1; <k> a.c / o1, o3 \} \]

Transitions from S11S32:
\[ T4 = \{ <k(k2)> \bar{a} \bar{c} \bar{d}; <k(k2)> \bar{a} \bar{c}.d; <k(k2)> \bar{a}.c.d / o3; <k(k2)> \bar{a}.c.d / o3 \} \]
\[ T5 = \{ <k(k2)> a \bar{c} \bar{d} / o1; <k(k2)> a \bar{c}.d / o1; <k(k2)> a.c \bar{d} / o1, o3; \]
\[ <k(k2)> a.c.d / o1, o3 \} \]
Theorem 3.3

When the refinement operator is applied on two deterministic and reactive FSMs the resulting FSM is deterministic and reactive.

Proof

mclk:
In the second version of the hierarchical operator (\(\downarrow\)) all mclk transitions are copied from the refined FSM. In the first version (\(\uparrow\)), this is the case in subsets 1 and 2 of \(T\). In subsets 3, 4 and 5, mclk transitions of the refined FSM are synchronously combined with mclk transitions of the refining FSM. It was already proved for Theorem 3.1 that synchronous product preserves reactivity and determinism.

Other clocks:
Transitions of the refined and refining FSMs remain unchanged. They are copied to subsets 6 and 7.

4.1.6 Mapping syntax to automata

We identify two syntactic domains, the set of gclk specifications \(PM\) and the set of non-gclk specifications \(PS\). Two special variables \(M\) and \(S\) will be used to stand for elements of \(PM\) and \(PS\) respectively. There are also two semantic domains discussed in the previous section, the set of gclk FSMs \(AM\) and the set of non-gclk FSMs \(AS\). Finally, two semantic functions, \(MSE: PM \rightarrow AM^{rd} \cup \{\bot\}\) and \(SSE: PS \rightarrow AS^{rd} \cup \{\bot\}\), are needed to map syntactic objects to semantic objects. \(AM^{rd}\) is the set of deterministic and reactive FSMs within \(AM\) while \(AS^{rd}\) is the set of deterministic and reactive FSMs within \(AS\). Specifications that do not meet the requirements for determinism and reactivity are considered incorrect and mapped to \(\bot\), pronounced “bottom”. Non-determinism and non-reactivity can be produced with the hiding operator.
Listed below are the production rules which show ways in which syntactic objects may be combined. On the syntactic side the synchronous parallel, asynchronous parallel, refinement and localization/hiding operations are labelled \( \bar{s} \), \( \bar{a} \), \( \bar{r} \) and \( \bar{l} \).

\[
M := M \bar{s} M | M \bar{a} S | M \bar{r}(q_1, R_1, q_2, R_2, \ldots, q_n, R_n) | M \bar{l}(a_1, a_2, \ldots, a_n)
\]

\[
S := S \bar{s} S | S \bar{r}(q_1, S_1, q_2, S_2, \ldots, q_n, S_n) | S \bar{l}(a_1, a_2, \ldots, a_n)
\]

\[
R := M | S
\]

All refinements can be specified in a single statement. On the other hand, the refinement operator is defined to work only with the refinement of a single state only. Therefore when giving the meaning to the refinement statement, the refinement operator needs to be applied repeatedly, as many times as there are refinements, in order to reach the final equivalent FSM. Note that subscript \( n \) in the refinement statement denotes the number of refined states, not the total number of states, since some states may be unrefined. The localization statement can also specify all local signals at once. Thus, as in the case of the refinement operator, the localization operator may need to be used multiple times to produce the FSM the localization statement is mapped to.

The syntax is mapped to semantics as follows:

\[
MSE(M_1 \bar{s} M_2) = \begin{cases} 
\bot & \text{if } MSE(M_1) = \bot \text{ or } MSE(M_2) = \bot \\
MSE(M_1) \| MSE(M_2) & \text{otherwise}
\end{cases}
\]

\[
MSE(M \bar{a} S) = \begin{cases} 
\bot & \text{if } MSE(M) = \bot \text{ or } SSE(S) = \bot \\
MSE(M) \\ \\ \| SSE(S) & \text{otherwise}
\end{cases}
\]

\[
MSE(M \bar{r}(q_1, R_1, q_2, R_2, \ldots, q_n, R_n)) = \begin{cases} 
\bot & \text{if } MSE(M) = \bot \text{ or } \exists i \in [1, n]: SE(R_i) = \bot \\
MSE(M) \downarrow_{q_i} MSE(R_i) & \text{if } R := M \\
MSE(M) \downarrow_{q_i} SSE(R_i) & \text{if } R := S \\
X_i = \begin{cases} 
X_{i-1} \downarrow_{q_i} MSE(R_i) & \text{if } R := M \\
X_{i-1} \downarrow_{q_i} SSE(R_i) & \text{if } R := S \\
\text{for } i \in [2, n]
\end{cases}
\end{cases}
\]
4.1.7 Integrating SDF graphs into automata semantics

Denotational semantics is the most common means of formally describing Kahn process networks. As a restriction of Kahn process networks, the family of dataflow process network models, which includes SDF, is also captured by denotational semantics. It would be very difficult to integrate denotationally described SDF graphs with operationally described synchronous FSMs. Therefore, we have to represent the SDFG as an FSM using the operators defined in previous sections. In every iteration the SDFG goes through two phases. In the first phase inputs are received and outputs from the previous iteration are sent. In the second phase inputs are processed and outputs are produced. The SDFG operation viewed as an FSM is formally defined below with the synchronous parallel and refinement operators.
Definition 4.9: Synchronous dataflow graph with m inputs and n outputs

\[ SDF = TOP \downarrow_{io} IOPAR \]

\[ TOP = (Q, q_0, I, R, O, V, C, T, RQ, Proc) \]

where

\[ Q = (io, processing), q_0 = io, \]
\[ I = \{gs\} \cup \{in\_qi\, |\, 1 \leq i \leq m\} \cup \{in\_qo\, |\, 1 \leq i \leq n\}, \quad R = \emptyset, \quad O = \emptyset, \]
\[ V = \{data\_in\, |\, 1 \leq i \leq m\} \cup \{data\_out\, |\, 1 \leq i \leq n\}, \quad C = \emptyset, \quad T = \{t_1, t_2\}, \]
\[ RQ = \emptyset, \quad Proc = \{assign\_output\_values\} \]

\[ t_1 = (sdflck, \emptyset, io, in\_qi, in\_qi, ... in\_qi_m, in\_qo, in\_qo, in\_qo, c, c, \emptyset, \emptyset, \{assign\_output\_values\}, io) \]

\[ IOPAR = IN_1 \parallel IN_2 \parallel ... \parallel IN_m \parallel OUT_1 \parallel OUT_2 \parallel ... \parallel OUT_n \]

\[ IN = (Q, q_0, I, R, O, V, C, T, RQ, Proc) \]

where

\[ Q = (q_1, q_2), \quad q_0 = q_1, \quad I = \{\emptyset\}, \quad R = \{ch\_in\}, \quad O = \emptyset, \quad V = \{data\_in\}, \quad T = \{t_1\}, \]
\[ RQ = \{(q_1, ch\_in)\}, \quad Proc = \{ch\_in\_proc\} \]

\[ t_1 = (sdflck, \emptyset, q_1, c, c, c, ch\_in, \emptyset, ch\_in\_proc, q_2) \]

\[ OUT = (Q, q_0, I, R, O, V, C, T, RQ, Proc) \]

where

\[ Q = (q_1, q_2), \quad q_0 = q_1, \quad I = \{\emptyset\}, \quad R = \{ch\_out\}, \quad O = \emptyset, \quad V = \{data\_out\}, \quad T = \{t_1\}, \]
\[ RQ = \{(q_1, ch\_out)\}, \quad Proc = \{ch\_out\_proc\} \]

\[ t_1 = (sdflck, \emptyset, q_1, c, c, c, ch\_out, \emptyset, ch\_out\_proc, q_2) \]
With this definition an SDF graph can be placed in the AS set (non-gclk FSMs) and used as the right operand of the asynchronous parallel operator. Figure 4.22 shows how an SDF graph with two inputs and two outputs is seen as an FSM. All transitions are driven by sdfclk. din and dout stand for data_in and data_out in definition 4.9. aov stands for assign_output_values. The two phases are represented as two states in the top level FSM. The io state is refined by concurrent FSMs that handle receiving and sending data. Note that the output FSMs send the initial values of their data_out variables before the first iteration. In the processing state data_in variables are read and data_out variables are written. Inputs of type in_qi or in_qo are present when their corresponding states are active. Gs is the signal that denotes that the outputs have been produced leading to the I/O state.

An important feature of “SDF FSMs” is that waiting for rendezvous can never be preempted on any channel. This significantly simplifies the implementation of rendezvous, which is described in Chapter 7 among other implementation aspects of DFCharts. The implementation of rendezvous is more difficult in a more general case where both sides are allowed to abort [109].
Obviously this is a very high level view as all internal signals are hidden inside the processing state. As we mentioned above what happens inside the processing state is usually described with denotational semantics. However, Kahn process networks and other dataflow models can also be described operationally as in [39], for example. When the operational semantics is employed, the whole dataflow network is typically seen as a labelled transition system where the state is determined by the state of processes and the state of channels. The state of a channel is simply the number of tokens present in it. In Kahn process networks and other dynamic dataflow models channels can grow infinitely large resulting in the infinite state space. On the other hand, channels are bounded in SDF. Furthermore, SDF processes contain no state; they transform data in the same way in each firing. Therefore the state of an SDF network is purely determined by the content of channels.

Figure 4.23 shows how an iteration of SDF1 in Figure 3.5 can be seen as a series of states. The first state is when two input tokens have been received in each input channel which makes actors A and B ready to fire. The second, third and fourth state are entered after actors A,B and C fire respectively. We need to emphasize that while an execution order has a finite state space, multiple execution orders exist for an SDF graph in general. Thus the processing state is not refined in the general definition, but it can be refined once the execution order has been decided.

Figure 4.23: Execution states of SDF1 from Fig. 3.5
4.2 TSM semantics

In TSM an event is defined as a pair \( e = (t, v) \), where \( t \in T \) is a tag while \( v \in V \) is a value. \( T \) is the set of tags while \( V \) is the set of values. A signal is a set of events. It is a member of \( 2^{T \times V} \) or a subset of \( T \times V \). It is defined as partial function from \( T \) to \( V \). Partial in this case means that a signal does not have to be defined for every \( t \in T \).

Tags do not necessarily represent physical time. They might only be used to denote the ordering of events. This is the case in DFCharts which is an untimed model. The order among events is described by the ordering relation “\( \leq \)”, which is defined on the set \( T \) [25]. The relation is reflexive \((\forall a \in T : a \leq a)\), antisymmetric \((\forall a, b \in T : (a \leq b \land b \leq a \rightarrow a = b))\), transitive \((\forall a, b, c \in T : (a \leq b \land b \leq c \rightarrow a \leq c))\). A related irreflexive relation is also defined, denoted “\( < \)”. The ordering of tags induces the ordering of events. Given two events \( e_1 = (t_1, v_1) \) and \( e_2 = (t_2, v_2) \), \( e_1 < e_2 \) if and only if \( t_1 < t_2 \). Two events can also have equivalent tags. In that case they are synchronous.

Since DFCharts is a heterogeneous model that consists of HCFSM and SDF models, we use two sets of tags \( T' \) and \( T'' \) in order to distinguish between the two constituent models. \( T' \) will be used to define synchronous signals in the HCFSM model, while \( T'' \) will be used to define asynchronous signals in the SDF model. \( T' \) is a totally ordered set which means that for any two tags \( t'_1 \) and \( t'_2 \), \( t'_1 \leq t'_2 \) or \( t'_1 < t'_2 \). \( T'' \) is a partially ordered set. This means that there exist two tags \( t''_1, t''_2 \in T'' \) such that \( t''_1 \not< t''_2 \).

We also define two sets of values \( V' \) and \( V'' \). \( V' \) is the set of values produced in the HCFSM (synchronous) domain while \( V'' \) is the set of values produced in the SDF (asynchronous) domain. An event from the synchronous domain is defined as \( e' = (t', vec) \) where \( t' \in T' \). As indicated in the previous section a valued event in DFCharts carries a group (vector) of values. For this reason we define \( vec \) as a tuple of \( n \) values which can be taken either from \( V' \) or \( V'' \). The interface processes which will be defined shortly, transport values across the two domains. In addition \( vec \) can take
two special values $\bot$ and $p$. $\bot$ denotes the absence of an event, while $p$ denotes that the event is present but it carries no value i.e. it is a pure event. The set of $n$-tuples over $V'$ is denoted as $V'^{n}$ while the set of $n$-tuples over $V^*$ is denoted as $V^*^{n}$. Thus $vec \in V'^{n} \cup V^*^{n} \cup \{\bot, p\}$.

Events in the asynchronous SDF domain carry only one value. An event in the asynchronous domain is defined as $e^* = (t^*, v)$, where $t^* \in T^*$ and $v \in V^* \cup V'$.

The communication between an SDF graph and FSMs is performed using two interface processes shown in Figures 4.24 and 4.25. STF converts SDF output signals into FSM input signals. FTS does the opposite. It converts FSM output signals into SDF input signals. Note that these processes are implicit as they are only used for semantic purposes. They are not seen in specifications such as that in Figure 3.5 where input and output signals in both directions are simply connected together. Both processes are triggered by the same tick signal that is used to trigger all FSMs in a DFCharts specification. When they consume inputs, they instantaneously produce outputs. Thus we can say that they belong to the synchronous domain of DFCharts. When triggered by a tick, each process invokes a set of functions that map parts of input signals to parts of output signals. Each function handles one output signal. Complete output signals are built as processes are repeatedly triggered by the tick signal.

![Figure 4.24: STF process](image)
4.2.1 Data transfer from SDF to FSM

STF has three inputs: the set of signals \( SO \), the set of signals \( RFI \) and the signal \( sor \). It outputs the set of signals \( FI \). The signals in \( SO \) and \( FI \) are explicit in specification while \( sor \) and the signals in \( RFI \) are implicit and should be generated during implementation.

We first describe the explicit signals.

\[ SO = \{ so_i | 1 \leq i \leq m \} \]

is the set of \( m \) output signals of the SDF graph that communicate with FSMs. When defined in the TSM framework each \( so_i \in SO \) is a set of events \( so_i = \{ e_{i,j}^s | j \in N \} \) where \( N \) is the set of natural numbers. \( e_{i,j}^s \in so_i \) is defined as \( e_{i,j}^s = (t_{i,j}^s, v_{i,j}) \), where \( t_{i,j}^s \in T^s \) and \( v_{i,j} \in V^s \) giving \( so_i \subseteq T^* \times V^* \). Events within \( so_i \) are totally ordered. \( j \) is an index that defines ordering. Given two events \( e_{i,p}^s \), \( e_{i,q}^s \), \( p < q \) (here \( < \) is the ordinary ordering relation for integers) implies \( e_{i,p}^s < e_{i,q}^s \).

However, as signals in \( SO \) are asynchronous, two events that belong to two different signals \( e_{x,p}^s \in so_x \) and \( e_{y,q}^s \in so_y \) are in general not related by \( \leq \), i.e. \( e_{x,p}^s \not\leq e_{y,q}^s \) regardless of \( p \) and \( q \).

\( FI \) is the set of synchronous input FSM signals that result from the conversion of the SDF output signals. Each \( so_i \in SO \) is converted into one \( fi_i \in FI \). Thus \( SO \) and \( FI \) are of the same size \( m \) we defined above i.e. \( FI = \{ fi_i | 1 \leq i \leq m \} \). Each \( fi_i \in FI \) is a set of events \( fi_i = \{ e_{i,j}^s' | j \in N \} \). \( e_{i,j}^s' \in fi_i \) is defined as \( e_{i,j}^s' = (t_{i,j}^s', v_{i,j}) \) where \( t_{i,j}^s' \in T' \) and
vec_{i,j} \in V^n \cup \{\perp\}$, hence $vec_{i,j} = (v_{i,j,1}', v_{i,j,2}', \ldots, v_{i,j,n_i}')$ carries $n_i$ values produced in the SDF domain i.e. $v_{i,j,1}', v_{i,j,2}', \ldots, v_{i,j,n_i}' \in V^*$ and it is possible that $vec_{i,j} = (\perp)$ which means that the event is absent. All events within the same signal $i$ have to carry the same number of values $n_i$. This is due to the rule that the SDF graph must produce the same number of output tokens in each iteration. Events in different signals can carry a different number of values. Within each $fi_i \in FI$ all events are totally ordered as in asynchronous SDF output signals. Moreover, events are also ordered across all $fi_i \in FI$. Given two events that belong to two different signals $e_{x,p}' \in fi_x$ and $e_{y,q}' \in fi_y$, $p < q$ implies that $e_{x,p}' < e_{y,q}'$. If $p = q$ then $e_{x,p}' = e_{y,q}'$ which means that $e_{x,p}'$ and $e_{y,q}'$ are synchronous events. Therefore we can drop the signal index in the tag so that $e_{i,j}' = (t_j, vec_{i,j})$.

The synchronous signals in $RFI$ indicate whether the input FSM channels are ready for rendezvous. In particular, each $rfi_i \in RFI$ shows whether the corresponding $fi_i$ is ready to receive data. Data can be sent through $fi_i$ if the rendezvous state that is connected to $fi_i$ is active. Each $rfi_i \in RFI$ is a set of events $rfi_i = \{e_{rf,j}' \mid j \in N\}$ where $e_{rf,j}' = (t_j', vec_{rf,i})$. The signals in $RFI$ are pure. Hence $vec_{rf,i}$ can take two values. If $rfi_i$ is present in the $j^{th}$ tick then $vec_{rf,i} = p$, otherwise if it is absent then $vec_{rf,i} = \perp$. The presence of $rfi_i$ indicates that $fi_i$ is ready to receive data whereas the absence indicates the opposite.

$sor$ (SDF outputs ready) also belongs to the synchronous domain. It is a set of events $sor = \{e_{sor,j}' \mid j \in N\}$, where $e_{sor,j}' = (t_j', vec_{sor,j})$ and $vec_{sor,j}$ can take one of the two values, $p$ or $\perp$. What the signals in $RFI$ do for the signals in $FI$, $sor$ does for the signals in $SO$. If $sor$ is present, the output SDF signals are ready to send data; if it is absent they are not ready. Since SDF output signals are all ready or not ready at the same time, only one signal is needed to indicate their status.

When $STF$ is triggered by a tick of the clock, it invokes the following function for each output channel:
\[ stf_i(e'_{\text{sor},j}, e'_{\text{rf},j}, \{e'_{i,1}, e'_{i,2}, \ldots, e'_{i,k_i+n_i}\}) = e'_{i,j} \]  

(4.1)

which is equivalent to

\[ stf_i((t'_{i,j}, \text{vec}_{\text{sor},j}), (t'_{i,j}, \text{vec}_{\text{rf},j}), \{(t'_{i,k_i+1}, v'_{i,k_i+1}), (t'_{i,k_i+2}, v'_{i,k_i+2}), \ldots, (t'_{i,k_i+n_i}, v'_{i,k_i+n_i})\}) = (t'_{i,j}, \text{vec}_{i,j}) \]  

(4.2)

with

\[
\text{vec}_{i,j} = \begin{cases} 
(v'_{i,k_i+1}, v'_{i,k_i+2}, \ldots, v'_{i,k_i+n_i}) & \text{if } \text{vec}_{\text{sor},j} = p \land \text{vec}_{\text{rf},j} = p \\
\bot & \text{otherwise}
\end{cases}
\]

where \( n_i \) is a constant that denotes the number of tokens (events in TSM) produced on the channel in one iteration of the SDF graph. On the other hand \( k_i \) is a variable which increases with the number of completed iterations. For example, if we assume that five tokens are produced in each iteration of the SDF graph, \( k_i = 0 \) after the first iteration and \( k_i = 5 \) after the second iteration. If we label the number of completed SDF iterations as \( r \) we can express \( k_i \) as

\[ k_i = r \cdot n_i \]  

(4.3)

If \( e'_{\text{sor},j} \) and \( e'_{\text{rf},j} \) are present \( stf_i \) takes values spread across multiple events in the asynchronous domain and groups them under a single event in the synchronous domain. Otherwise if \( e'_{\text{sor},j} \) is absent (SDF graph is not ready to communicate) and/or \( e'_{\text{rf},j} \) is absent (the rendezvous state for \( j \) is not active) then the communication does not occur so \( e'_{i,j} \) is absent.

An important effect of \( STF \) is that values carried by asynchronous signals in \( SO \) can become synchronous as shown in the example below where there are three channels and five iterations have been completed.
Each event is defined as a set of synchronous FSM signals that present and they will carry the value from the asynchronous domain.

4.2.2 Data transfer from FSM to SDF

FTS performs conversion in the opposite direction. It takes three inputs: the set of signals \(FO\), the set of signals \(RFO\) and signal \(sir\). It outputs the set of signals \(SI\). \(FO\) is the set of synchronous FSM signals that provide input data for the SDF graph. We define that there are \(m\) signals in \(FO\) so that \(FO = \{fo_i \mid 1 \leq i \leq m\}\). Each \(fo_i \subseteq FO\) is a set of events \(fo_i = \{e'_{i,j} \mid j \in N\}\). \(e'_{i,j} \in fo_i\) is defined as \(e'_{i,j} = (t'_j, vec_{i,j})\) where \(t'_j \in T'\) and \(vec_{i,j} \in V'^{T'_i} \cup \{\perp\}\). Thus \(fo_i \subseteq T' \times (V'^{T'_i} \cup \{\perp\})\) and \(vec_{i,j} = (v'_i, v'_j, \ldots, v'_{i,j,d})\) are \(l_i\) values produced in the synchronous domain i.e. \(v'_{i,j,1}, v'_{i,j,2}, \ldots, v'_{i,j,d} \in V'\). If the event is absent then \(vec_{i,j} = (\perp)\). \(SI\) is the set of SDF input signals that receive tokens from FSMs. As each \(fo_i \subseteq FO\) is converted into one \(si_i \in SI\), there are \(m\) signals in \(SI\) i.e. \(SI = \{si_i \mid 1 \leq i \leq m\}\). Each \(si_i \in SI\) is a set of events \(si_i = \{e''_{i,j} \mid j \in N\}\). \(e''_{i,j} \in si_i\) is defined as \(e''_{i,j} = (t''_j, v_{i,j})\) where \(t''_j \in T''\) and \(v_{i,j} \in V''\) giving \(si_j \subseteq T'' \times V''\). Ordering of events in synchronous HCFSM and asynchronous SDF signals was already described when \(STF\) was discussed.

Each \(rfo_j \in RFO\) is a set of events \(rfo_j = \{e'_{rfo,j} \mid j \in N\}\) where \(e'_{rfo,j} = (t'_j, vec_{rfo,j})\) and \(vec_{rfo,j} \in \{\perp, p\}\). If \(e'_{rfo,j}\) is present the corresponding rendezvous state is active, if it is absent the state is inactive. On the other hand \(sir = \{e'_{sir,j} \mid j \in N\}\) which also belongs to the synchronous domain, shows the status of the SDF input signals. If \(e'_{sir,j}\) is present the SDF graph is ready to receive inputs. If \(e'_{sir,j}\) is absent it is not.
When FTS is triggered by a tick of the clock, it invokes the following function for each output channel:

\[
fts_i(e'_{sir,j}, e'_{rf,j}, e'_{i,j}) = \begin{cases} \\
\{e^*_{i,q_i+1}, e^*_{i,q_i+2}, \ldots, e^*_{i,q_i+l_i}\} \quad \text{if } vec_{sir,j} = p \wedge vec_{rf,j} = p \\n\emptyset \quad \text{otherwise}
\end{cases}
\]

where

\[
\{e^*_{i,q_i+1}, e^*_{i,q_i+2}, \ldots, e^*_{i,q_i+l_i}\} = \{(t^*_{i,q_i+1}, v'_{i,j,1}), (t^*_{i,q_i+2}, v'_{i,j,2}), \ldots, (t^*_{i,q_i+l_i}, v'_{i,j,l_i})\}
\]

In (4.4) \(l_i\) is a constant that denotes the number of tokens consumed on \(s_i\) in one SDF iteration while \(q_i\) is a variable which can be expressed as

\[
q_i = r \cdot l_i
\]

where \(r\) is the number of completed SDF iterations.

An important effect of FTS is that values carried by synchronous signals in FO become desynchronised as shown in the example below where there are three channels and five iterations have been completed. We also assume that all input events are present.

\[
\begin{align*}
fts_1(e'_{sir,j}, e'_{rf,j}, e'_{1,j}) & = \{e^*_{1,5_l+1}, e^*_{1,5_l+2}, \ldots, e^*_{1,5_l+l_i}\} \\
fts_2(e'_{sir,j}, e'_{rf,j}, e'_{2,j}) & = \{e^*_{2,5_l+1}, e^*_{2,5_l+2}, \ldots, e^*_{2,5_l+l_i}\} \\
fts_3(e'_{sir,j}, e'_{rf,j}, e'_{3,j}) & = \{e^*_{3,5_l+1}, e^*_{3,5_l+2}, \ldots, e^*_{3,5_l+l_i}\}
\end{align*}
\]

### 4.3 The impact of clock speeds

In section 4.1.1, when we discussed determinism we focused on each clock domain separately without paying attention to effects caused by the relation between clocks. The aim of this section is to show how clock speeds can impact the overall behaviour of
a system. For some systems such as the frequency relay, clock speeds are only important for time constraints. For others such as the one presented in Figure 4.26, they completely change the system behaviour.

The specification in Figure 4.26 does not have any input signals but it has two output signals \( c \) and \( d \). When FSM1 makes a transition from S3 to S1 both SDF1 and FSM3 are ready to operate. After each iteration SDF1 outputs integer 3 on ch1 while ch2 is only used for synchronization. FSM3 increments shared variable \( v2 \) in every tick.

Both of its transitions are always enabled. If SDF1 always takes three gclk ticks to complete its iteration, the system will not produce any output. If it always takes more than three ticks a sequence of \( c \) outputs will be produced and if it always takes less than three ticks a sequence of \( d \) outputs will be produced. In addition, any combination of
behaviours is possible if the length of SDF1 iterations is not constant in terms of $gclk$ ticks.

Determinism is not viewed consistently across the literature. In [110], the authors claim that Multiclock Esterel is deterministic even though the behaviour of a specification can be influenced by clock speeds as in DFCharts. The non-determinism arising from clocks is considered to be external. On the other hand, the Kahn process networks (KPN) model is said to be deterministic exactly because the output sequence is independent of process speeds. Whether we call non-determinism due to clocks external or internal it does pose a problem to design if not handled properly. We could completely avoid this issue in DFCharts semantics by fixing the speed of every SDFG to a constant number of $gclk$ ticks. That would be a poor solution because it would severely reduce implementation space. Instead, we allow experimenting with different speeds of SDF graphs relative to $gclk$ in DFCharts design flow. As a result verification becomes more intensive but efficient implementation can be obtained in the end.
Chapter 5

DFCharts in SystemC and Esterel

With graphical syntax presented in Chapter 1 and Java-based textual syntax given in Chapter 6, DFCharts can be used as a language for specification of embedded systems. In this chapter we use DFCharts as a model of computation to assess the effectiveness of two popular system level languages, SystemC and Esterel, in capturing behaviour of heterogeneous embedded systems. While SystemC is being proposed by an industry consortium and has no formal semantics, Esterel has a formal semantics and formal verification capabilities. Hence, both these languages represent differing perspectives on system-level modelling. The frequency relay case study was specified in both languages, following the DFCharts model as closely as possible. Using this analysis, we can identify what needs to be improved in each language in order to increase their ability to handle heterogeneous embedded systems. The relation between the two languages and DFCharts was previously described in [111].

In section 5.1 we list the requirements for SystemC and Esterel according to DFCharts features. For each requirement we discuss the level of support provided by SystemC and Esterel mainly by using observations made while specifying the frequency relay in both languages. As neither language was able to completely follow the target DFCharts model, we describe the resulting deviations from DFCharts in the frequency relay specifications. In section 5.2, we discuss some numerical results, such as code size and
simulation speed, that were obtained after describing the frequency relay in SystemC and Esterel. Finally, section 5.3 suggests some modifications in the semantics and syntax of SystemC and Esterel that would lead to their better support for heterogeneous embedded systems.

5.1 Analysis based on requirements

The list of requirements is the following:

1. Concurrent processes – an essential requirement, which is a precondition for all the other points to follow.
2. Rendezvous communication.
3. Buffered communication between processes and specification of firing rules for dataflow modules.
4. Support for HCFSM model with synchronous communication.
5. Imperative statements to describe data transformations inside SDF actors, as well as smaller computations performed by FSMs.
6. Hierarchy and preemption - multiple processes inside a hierarchical state and termination of lower-level processes instantly upon an exit from the hierarchical state.

5.1.1 Concurrent processes

SystemC

Concurrency is implicitly assumed in SystemC. Processes defined in a single module are concurrent. When multiple modules are connected at any level of hierarchy they always execute concurrently. In fact, specifying the execution order of modules such as sequential or pipelined execution available in other languages like SpecC [79], is not possible in SystemC. The designer would have to manipulate the execution order of modules by using control signals.
Concurrency can be explicitly created using the parallel operator $\parallel$ at any level of hierarchy. The branches in a parallel operator can contain any number of statements. The $\parallel$ operator creates concurrent threads that communicate and synchronize using synchronous broadcast. This is based on the SR model of computation where a global clock is assumed. Inputs and corresponding outputs are generated in the same tick of the global clock, leading to a zero delay model. Also, events generated in any thread are broadcasted to all other threads. Any other form of concurrency will have to be emulated by clever programming.

**5.1.2 Rendezvous communication**

*SystemC*

SystemC does not offer any high level construct that implements rendezvous directly. However it should not be difficult to create rendezvous between two processes using `wait` statements.

*Esterel*

As in SystemC rendezvous cannot be specified directly. It has to be created using a combination of `await` and `emit` statements that need to be programmed appropriately.

**5.1.3 Buffers and firing rules for dataflow**

*SystemC*

FIFO buffers can be implemented in SystemC by the primitive channel `sc_fifo`. Because of constant data rates, SDF blocks should be implemented as method processes. There should be no need to use less efficient thread processes. However only thread processes that are dynamically scheduled by the SystemC kernel can use `sc_fifo` buffers. As a result, static scheduling of the SDF model cannot be easily implemented. Moreover,
firing rules associated with the SDF actors are not supported by the primitive FIFO channel. The SystemC kernel can activate a thread process as soon as data is available in its input FIFO channel. Hence, the third requirement of buffered communication with static scheduling can not be directly implemented using a SystemC construct.

Due to the reasons explained above, the three data-dominated blocks were implemented as thread processes that communicate through sc_fifo channels using blocking reads and blocking writes (blocking writes are inevitable as sc_fifo channels must be bounded). Thus the three signal processing blocks appear as a Kahn process network in the SystemC specification rather than an SDF network.

**Esterel**

A FIFO buffer can be implemented as a separate process (C functions) in Esterel. In this way computation and communication would be separated which may be a useful feature if components need to be reused. However, the FIFO process would still be synchronized to the tick signal. Thus the level of abstraction would be lower than in asynchronous SDF buffers. Alternatively, buffers could be implemented as **asynchronous tasks** but this would lead to integration problems which will be discussed below.

It is clear that Esterel cannot efficiently capture the data-dominated (SDF) part of the frequency relay. Therefore, the SDF blocks performing signal processing (averaging filter, symmetry function and peak detection) must be reactive as all other processes in the system. The event they react to is a sample coming from the analogue-to-digital converter. The problem comes from the fact that all processes must be aligned to a single tick signal i.e. they have to read inputs and produce outputs at the same time. The most efficient solution for the SDF processes is that the tick signal coincides with the sampling frequency of the AC input signal. On the other hand this would be too slow for the parameter settings block whose inputs may be faster than the sampling frequency especially in the scenario when it is connected to a high-speed CDMA network. The ticks must be frequent enough to capture all inputs in the system. Thus the rate of the tick signal is determined by the process with the fastest inputs in the system, which is
the parameter settings block. The consequence is an implementation which is likely to be inefficient since the data-dominated blocks have to make their computations faster than they need to. A more efficient implementation would be achieved if the data-dominated blocks were specified as asynchronous tasks taking more than one tick to complete computations. The problem with asynchronous tasks is in that they are not handled by Esterel analysis tools. In Esterel Studio, the design environment based around Esterel, every asynchronous task is seen as a black box and has to be simulated in another environment. Return times of asynchronous tasks and values they return have to be entered in the same way as inputs in Esterel Studio simulations. For these reasons, in order to be able to make a complete simulation in Esterel Studio, we had to make the data-dominated blocks behave according to the synchrony hypothesis.

Multiclock Esterel [58] was proposed to alleviate some of the problems described above. In this extension of Esterel processes may run at different speeds, which implies that there are multiple tick signals in the system. Processes that are triggered by different ticks can use various mechanisms to synchronize including sampling and reclocking [110].

5.1.4 HCFSM with synchronous/reactive communication

SystemC

In SystemC an FSM can be described with a switch-case construct, which can be nested to describe hierarchical FSMs. This involves using multiple state variables. Figure 5.1 shows a section of the method process that describes the parameter settings and threshold reception FSMs of the frequency relay. The protocol for receiving thresholds starts from state s2_0, which is nested inside state s2. Every time state s2 is entered signals cancel and done are checked before the execution of the threshold reception protocol. The description in Figure 5.1 is essentially behavioural level abstraction. It is also cycle-accurate since the process is driven by a clock.

Simultaneous events, that are necessary for synchronous communication between hierarchical FSMs, can be created in SystemC. However, instantaneous communication
is not handled in the same way as in synchronous languages. The SystemC kernel operates according to the discrete event model where simultaneous events are resolved with microsteps. In contrast, a fixed point solution is sought in the semantics of synchronous languages.

Reactivity is supported by signal sensitivities and wait statement. In control-dominated systems, a consequence of a reaction is often a pre-emption. SystemC lacks powerful preemption constructs such as abort and trap. Therefore, it does not fully satisfy the fourth requirement completely.

```
case s2:
    if (cancel) {
        state = s0;
        substate = s2_0;
    }
    else if (done) {
        state = s3;
        substate = s2_0;
    }
    else
        switch (substate) {
            case s2_0:
                if (skip) {
                    skip = false;
                    substate = s2_1;
                    thresh_code = 2;
                }
                else if....
```

Figure 5.1: Hierarchical FSMs in SystemC

Esterel

Obviously, the synchronous reactive model is completely supported by Esterel. Esterel contains plenty of statements that enable specifying control-dominated behaviour in a natural way. For example, reactivity is supported by statements such as await. Pre-emption can be naturally described by statements such as abort and trap. This is illustrated in Figure 5.2, which represents Esterel code for the timer in the frequency relay. When the counting of 8000 ticks is finished, the block emits signal time out. However, the operation can be preempted by signal start causing the counter to reset.
await start;
loop
  var counter := 0 : integer in
  weak abort
    positive repeat 8000 times
      await tick;
      counter := counter + 1
    end repeat;
  emit time_out;
  halt
end when start
end var
end loop

Figure 5.2: Esterel specification of timer in frequency relay

All Esterel specifications including those that contain pre-emption statements can be translated into an FSM. The benefit of using pre-emption statements such as *abort* and *trap* are potentially many states and transitions that are implicit. As a result the specification becomes compact and easily readable. However, there are many situations when a designer wants all states of an FSM to be explicitly represented in the specification. This is where Synccharts, a Statecharts version based on the Esterel semantics, complements Esterel.

### 5.1.5 Data transformations

*SystemC*

As an imperative language, SystemC provides an excellent support for describing sequential algorithms and thus it satisfies the fourth requirement quite well. To illustrate this point, a section of the code from the main loop of the averaging filter (a thread process) is shown in Figure 5.3. The main operation is contained inside the *else branch*. The algorithm reads a sample, performs averaging inside the loop and then the result is written. The control signal *measure_off* is checked at the beginning of the loop. The reason for the presence of this signal will be explained when the next requirement is discussed.
while(1) {
    if (measure_off.read()) {
        ...........
    }
} else {
    index = (index+1)%ws;
    window[index] = sample_in.read();
    float sum = 0;
    for (j=0;j<ws;j++)
        sum = sum + window[(index+j)%ws];
    ave_result.write(sum/ws); // output average
}  

Figure 5.3: Section of SystemC code for averaging filter in frequency relay

Esterel

C is available as a host language in Esterel and hence complex algorithms for data transformations inside data-dominated blocks can be specified in a similar way as in Figure 5.3. As already discussed, the problem is the assumption of instantaneous computation that has to be applied to time-consuming algorithms. Otherwise, they are out of reach of analysis tools in Esterel Studio when modelled as asynchronous tasks.

5.1.6 Multiple processes inside a state

SystemC

Exception handling is a key component of the frequency relay behaviour as modelled by the top level FSM in Figure 3.6. When events off or reset occur, state S2 should be left and all processes in it instantly terminated. SystemC does not fulfil this requirement since there is no direct way to implement exceptions modelled by exits from higher-level hierarchical states. It was indicated earlier that hierarchy in an FSM could be modelled by nested switch-case statements; this type of modelling is not applicable here since it is not possible to instantiate processes inside a case branch.

As the top level FSM in Figure 3.6 cannot be directly implemented, the execution of each process has to be controlled by one or more control signals. Due to this, a separate
FSM named “global state controller” was added to enable suspension and reactivation of the FSMs and SDFG. It resides with other processes at the same hierarchical level. The control signal measure_off in Fig. 5.3 comes from this FSM. If measure_off is active, the process does not execute the main operation. It becomes active only when measure_off is deactivated.

A modified model of the frequency relay was created to better suit SystemC and this is shown in Figure 5.4. As already mentioned, the three SDF actors are implemented as threads and cannot be terminated instantaneously. Each of them can change its state from active to inactive or vice versa only after reading a control signal, which is done before the beginning of each iteration. This is in contrast to the original model in Figure 3.6, which assumes the SDF graph and all FSMs in state S2 can be terminated instantaneously.

![Diagram of SDF1 – find peaks](image)

<table>
<thead>
<tr>
<th>FSM8 global state controller</th>
<th>FSM2 parameter settings</th>
<th>FSM6 timer</th>
<th>FSM5 switch control</th>
<th>FSM4 rate of change calculation</th>
<th>FSM3 frequency calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.4: Modified model to better suite the SystemC specification

_Esterel_

The sixth requirement is fully satisfied by Esterel. It supports behavioural hierarchy and has several statements that enable preemption. For example, concurrent blocks can be run inside the abort statement.
5.1.7 Comparison between SystemC and Esterel

<table>
<thead>
<tr>
<th>Requirement</th>
<th>SystemC</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Concurrent processes</td>
<td>xxx</td>
<td>xxx</td>
</tr>
<tr>
<td>2 Rendezvous communication</td>
<td>xx</td>
<td>xx</td>
</tr>
<tr>
<td>3 Dataflow</td>
<td>xx</td>
<td></td>
</tr>
<tr>
<td>4 HCFSM</td>
<td>xx</td>
<td>xxx</td>
</tr>
<tr>
<td>5 Data transformations</td>
<td>xxx</td>
<td>xxx</td>
</tr>
<tr>
<td>6 Multiple processes in a state</td>
<td></td>
<td>xxx</td>
</tr>
</tbody>
</table>

Table 5.1: Level of support provided by SystemC and Esterel

The requirements are listed again in Table 5.1 which also shows the level of support (in a scale of 0 to 3 as indicated by the number of Xs) for a given feature in SystemC and Esterel. Both languages fully support the requirements one and five. It is also obvious from the previous discussion that SystemC has no support for the sixth requirement while Esterel fully satisfies it. On the other hand, Esterel has no support for dataflow. Although no direct support is available for rendezvous, it can be constructed in both languages. SystemC does not fully satisfy the third requirement since only thread processes can be used with FIFO channels. As a result, only dynamic scheduling is possible. Furthermore, there is no way to directly specify firing rules in SystemC. The fourth requirement is also not fully supported by SystemC due to the lack of pre-emption.

5.2 Numerical results

The files that comprise the SystemC specification of the frequency relay are shown in Table 5.2 with the number of lines of code for each file. Each file represents a process from the model in Figure 5.4. The exception is frequency Relay.cpp, the top level file where all lower level processes are connected. In addition, this file also contains the code that specifies the global state controller. At the bottom of the table is the testbench...
file, which generates input stimuli for the frequency relay and records outputs in a .vcd file.

The Esterel specification was created in Esterel Studio. Table 5.3 shows the list of files and their sizes. This specification is a mixture of Esterel files whose extension is .strl and C files whose extension is .c. In this case, the file structure does not completely follow the model in Figure 3.6 mainly because of the fact that two languages had to be used. Dataflow.strl represents the three SDFG blocks. It invokes averaging_filter.c and symmetry_function.c, while the peak detection algorithm is described with Esterel statements. Most intensive computations appear inside these two C files, not dataflow.strl. Measurement.strl implements the frequency calculation and rate of change calculation blocks. It uses two C files freq_average.c and roc_average.c for minor computations. Switch_control.strl implements both the switch control and timer blocks. Frequency_relay.strl is the top level file, which connects all lower level blocks but it also implements the top level FSM. As we are only concentrating on syntax for the moment, it is irrelevant whether the C files in Table 5.3 represent instantaneous functions or asynchronous tasks.

<table>
<thead>
<tr>
<th>System C files</th>
<th>Code size (number of lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>averaging_filter.cpp</td>
<td>85</td>
</tr>
<tr>
<td>symmetry_function.cpp</td>
<td>95</td>
</tr>
<tr>
<td>peak_detection.cpp</td>
<td>66</td>
</tr>
<tr>
<td>frequency_calculation.cpp</td>
<td>93</td>
</tr>
<tr>
<td>roc_calculation.cpp</td>
<td>100</td>
</tr>
<tr>
<td>parameter_settings.cpp</td>
<td>239</td>
</tr>
<tr>
<td>switch_control.cpp</td>
<td>135</td>
</tr>
<tr>
<td>timer.cpp</td>
<td>38</td>
</tr>
<tr>
<td>frequency_relay.cpp</td>
<td>251</td>
</tr>
<tr>
<td>testbench.cpp</td>
<td>412</td>
</tr>
</tbody>
</table>

Table 5.2: SystemC files for frequency relay specification
The total code size for the SystemC specification excluding the testbench file is 1102 lines while the total code size for the Esterel specification is 901 lines. This difference is not significant taking into account that the SystemC specification had more files and thus more declarations. It should also be noted that the declarations of C functions used by the Esterel specification are contained in separate Esterel Studio files not listed in the table.

Many different input patterns had to be specified in order to make a satisfactory simulation for the frequency relay. It is no surprise that the testbench is the biggest file in Table 5.2. While both the testbench and system under test can be specified in SystemC, only the actual system can be specified in Esterel. In Esterel Studio it is possible to run step-by-step interactive simulations. While this feature is useful for quick checks, it would be inefficient for the complete simulation of a large system. Instead all inputs have to be created in advance. For that purpose, Esterel Studio offers scenario files. However, writing scenario files manually is tedious since a very simple language has to be used. It is probably faster to generate a scenario file as an output of a program written in a high level programming language. All this still takes more time than writing a testbench in SystemC.

<table>
<thead>
<tr>
<th>Esterel files</th>
<th>Code size (number of lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow.strl</td>
<td>76</td>
</tr>
<tr>
<td>averaging_filter.c</td>
<td>34</td>
</tr>
<tr>
<td>symmetry_function.c</td>
<td>41</td>
</tr>
<tr>
<td>measurement.strl</td>
<td>77</td>
</tr>
<tr>
<td>freq_average.c</td>
<td>31</td>
</tr>
<tr>
<td>roc_average.c</td>
<td>43</td>
</tr>
<tr>
<td>parameter_settings.strl</td>
<td>251</td>
</tr>
<tr>
<td>switch_control.strl</td>
<td>139</td>
</tr>
<tr>
<td>frequency_relay.strl</td>
<td>209</td>
</tr>
</tbody>
</table>

Table 5.3: Esterel files for frequency relay specification
While the time it takes to prepare a simulation is important, a more important factor to consider is the actual simulation time. It took close to four hours for the simulation of the Esterel specification versus just five minutes for the simulation of the SystemC specification. Both simulations were done on the same computer. The SystemC simulation was done in Microsoft Visual C++ version 6 with SystemC class library 2.0.1. The Esterel simulation was done in Esterel Studio version 4, which supports Esterel v5.

There are probably several factors that caused such huge difference but the most interesting one is to do with the modelling in Esterel. The whole system has to run on one clock since Esterel does not support multiple clocks. The speed of the system is determined by the process with the fastest changing inputs, which is the parameter setting block. This speed is unnecessarily high for data-dominated parts which need to read inputs only when a sample arrives. As a consequence, there are many ticks with absent inputs in this part of the system.

It should be noted that Esterel model is first converted into C code and then simulated. It is quite possible that conversion into C code is still not as efficient as it could be. This can also have a large impact on the simulation time.

Even though SystemC code can be simulated much faster than Esterel code, we cannot state that in general SystemC has better validation capabilities than Esterel. Although simulation is the most widely used method of validation, it is not the only one. The other method is formal verification, which may be applied to Esterel specifications (unlike SystemC). In the frequency relay case study, formal verification has limited use since any useful properties that could be verified would be related to data-dependant internal activities rather than inputs and outputs. It would be difficult to define such properties using Esterel observers that only check properties over the control part.
5.3 Feature extensions of SystemC and Esterel

According to the analysis presented in section 5.1, SystemC could better support or does not support at all requirements 2, 3, 4 and 6. A rendezvous channel can be constructed by a designer using wait and notify statements to create request and acknowledge lines that are necessary for the rendezvous protocol, although that could take some effort. Ideally, a standard rendezvous channel should be added to the library of channels that includes sc_fifo, sc_signal etc. Asynchronous thread processes that communicate through FIFO channels using blocking reads provide a good foundation for dataflow models. On the other hand, it is still difficult to specify firing rules and construct static scheduling orders. Improvements need to be made in that direction. Synchronous processes can be created in SystemC, which is essential for the fourth requirement. Reactivity can also be modelled using signal sensitivities. On the other hand, the non-existence of preemption is a serious disadvantage when modelling control-dominated behaviour. Processes cannot be instantaneously terminated or interrupted, which is necessary for the sixth requirement. This fundamental limitation can be overcome only by making deep changes in the simulation semantics of SystemC.

SystemC-H [112] is an extension of SystemC, where some desired changes discussed above have already been implemented. The SystemC kernel was extended to provide better support for SDF, CSP and FSM models. In SystemC-H it is possible to construct static schedules for SDF models, which leads to increased simulation efficiency compared to SystemC. Another important addition is hierarchical heterogeneity described in [113], which makes it possible to refine a state of an FSM with an SDF graph. In its current form, SystemC-H would probably not be able to support DFCharts entirely since it adheres to purely hierarchical heterogeneity found in Ptolemy. DFCharts represents a mix of hierarchical and parallel heterogeneity.

Like SystemC, Esterel does not directly support rendezvous, but it is possible to construct it using await and emit statements. The main problem of Esterel is a complete lack of support for requirement 3. The assumption made by the synchrony hypothesis that all computations are instantaneous is often not valid for data-dominated systems.
Furthermore, Esterel syntax is not appropriate for dataflow. It would be possible to design a dataflow network inside an asynchronous task. However, the asynchronous task is currently just a black box in Esterel’s semantics. Describing something in an asynchronous task means going outside Esterel and its development tools. In order to create a solid basis for an integrated environment, it is necessary to define a model of computation for asynchronous tasks, which could be SDF, and to interface it with the synchronous reactive model.
Chapter 6

Java environment

This chapter presents a Java environment for DFCharts based design. The specification of finite state machines, synchronous dataflow graphs and communication mechanisms used in DFCharts is supported by a Java class library. Besides, the DFCharts library, Ptolemy is needed for simulation of SDF graphs.

A DFCharts specification is created in Java by describing FSMs and SDFGs and connecting them. All FSMs and SDFGs used in a specification have to be defined as classes and then instantiated at the appropriate place in the specification hierarchy. We will describe FSM and SDFG classes in sections 6.1 and 6.2 using examples from the frequency relay specification. Besides FSM and SDFG classes, a DFCharts specification also needs a class that handles the top level. This type of class will be described in section 6.3. Section 6.4 describes how DFCharts specifications are executed in Java. While the first four sections deal with specification and simulation in Java from the designer’s perspective, section 6.5 describes the class library that enables this process. The chapter is concluded in section 6.6 by looking at differences between the frequency relay specification in Java and those in SystemC and Esterel.

What is described in the following sections can also be used as the basis for a graphical user interface so that DFCharts specifications would look exactly like the figures in Chapter 3. Java provides a wide range of utilities for graphical programming especially
in the *swing* package. Graphical objects would need to be converted into instances of FSM and SDFG classes, which should be a straightforward process.

### 6.1 FSM classes

Figure 6.1 shows the structure of an FSM class. Each FSM class has to extend *BaseFSM*, a library class. As seen from the figure, the constructor plays a major role in the class definition since it contains several code sections. Each code section will be described below. The modifier *public* is optional. It is not needed if the class does not need to be seen outside its package.

```java
public class FSM extends BaseFSM {

    reference variables for input signals
    reference variables for output signals
    reference variables for shared variables
    reference variables for states
    reference variables for local variables

    inner classes for transition inputs
    inner classes for transition outputs

    FSM (inputs, outputs, shared variables, channels...) {       // constructor
        base constructor parameters

        input signal connections
        output signal connections
        shared variable connections
        local variable initializations

        instantiation of states
        instantiation of transition inputs and outputs
    }
}
```
state connections

instantiation of local signals for lower level FSMs
instantiation of shared variables for lower level FSMs
instantiation of channels for lower level FSMs and SDFGs

instantiation of lower level FSMs
instantiation of lower level SDFGs

state refinements

Figure 6.1: Structure of FSM classes

6.1.1 Declaration of reference variables for I/O signals, states and variables

DFCharts library classes *InputSignal, OutputSignal, HierarchicalState, SimpleState, SharedVariable* represent input signals, output signals, simple states, hierarchical states, and shared variables respectively. Figure 6.2 shows the part of class FSM4 (rate of change calculation) which declares variables for these data types.

```c
InputSignal start_roc;
OutputSignal rd;
SharedVariable<Integer> roc_stat;
SharedVariable<Float> ave_freq;
SharedVariable<Float> roc_thresh1;
SharedVariable<Float> roc_thresh2;
SharedVariable<Float> roc_thresh3;
SimpleState s41, s42, s43;
float first, second;    // last two values of ave. freq. used for roc calculation
float temp;
float roc, ave_roc;
int roc_ws;            // buffer size
float[] roc_window;    // buffer containing rate of change samples
```

Figure 6.2: Signals, variables and states in FSM4
In the frequency relay specification in section 3.2, \(rd\) and \(start\_roc\) appear as local signals, but from the point of view of FSM4, they are seen as input and output signals. Their role as local signals becomes apparent when FSM1 is constructed. This detail will be shown later in the section. While input and output signals are pure in DFCharts, shared variables are associated with data types. Currently only primitive data types are supported such as float, integer, boolean. They have to be used in the form of wrappers. For example, Float is written instead of float. On the other hand, local variables can be of any data type. The local variables of FSM4, which are declared after the states, all have primitive data types.

### 6.1.2 Inner classes for transition inputs and transition outputs

In the general transition form \(t[c]/O,P\), the part before the slash is called *transition input* while the part after the slash is called *transition output*. A transition input consists of signals and variable conditions. In Java, it is defined as a class that extends the DFCharts library class called *TransitionInput*. *TransitionInput* is an abstract class that contains the single abstract method called evaluateInput(). Thus a class that extends *TransitionInput* has to have a method that implements evaluateInput(). Figure 6.3 shows the class that specifies the input for the transition that leads from S41 to S42 in FSM4.

```java
class s41input1 extends TransitionInput {
    public boolean evaluateInput() {
        if (start_roc.read()) {
            return true;
        } else {
            return false;
        }
    }
}
```

Figure 6.3: The input for transition S41 \(\rightarrow\) S42 in FSM4

InputSignal has the read() method which returns true if the signal is present. The return type of evaluateInput() is boolean, so either true or false must be returned.

A transition output can contain emitted signals and procedures. It is defined as a class that extends the DFCharts library class called *TransitionOutput*. *TransitionOutput* is an
abstract class with the single abstract method called computeOutput(). All classes that extend it must implement this method. Figure 6.4 shows the class that specifies the output for the transition that leads from S43 to S41 in FSM4. Another example is given in Figure 6.5, which is the output of the transition from S42 to S43.

```java
class s43output1 extends TransitionOutput {
    public void computeOutput() {
        rd.write(true);
    }
}
```

Figure 6.4: The output for transition S43 → S41 in FSM4

```java
class s42output1 extends TransitionOutput {
    public void computeOutput() {
        float rt1, rt2, rt3;
        rt1 = roc_thresh1.read();
        rt2 = roc_thresh2.read();
        rt3 = roc_thresh3.read();
        if (ave_roc < rt1)
            roc_stat.write(0);
        else if (ave_roc >= rt1 && ave_roc < rt2)
            roc_stat.write(1);
        else if (ave_roc >= rt2 && ave_roc < rt3)
            roc_stat.write(2);
        else if (ave_roc >= rt3)
            roc_stat.write(3);
    }
}
```

Figure 6.5: The output for transition S42 → S43 in FSM4

Emitting output signals and computing procedures are both done inside computeOutput() method. In Figure 6.4, just one output is emitted by passing true to write() method of OutputSignal. The use of SharedVariable class is illustrated in Figure 6.5. It can be noticed that the value of a SharedVariable object is read using read() method and written using write() method.

It may seem unnecessary that transition inputs and outputs have to be specified as classes. It would be easier to simply create methods that are not inside classes. However, BaseFSM class in the DFCharts library, that all FSMs have to inherit from,
would not be able to handle execution in that case. The reason is the fact that a method in Java can only be invoked with its name. It is not possible to create a variable that points to a method. On the other hand, BaseFSM is able to call transition inputs and outputs, which implement abstract classes.

### 6.1.3 Constructor parameters

The parameters passed to the constructor of an FSM include references for input signals, output signals, shared variables and internal channels. In addition, the name is needed as well as the reference for the refined (parent) FSM. It is also possible for an FSM to be instantiated at the top level. In that case the reference for the top level class is passed to the constructor.

The base constructor has to be handled as well. It takes the reference to the object that instantiates the FSM (another FSM or top level), the FSM name, the number of states, and references to output signals.

All input and output signals are created at the top level. A reference to a SharedVariable object may be passed to an FSM, but an FSM can also create a shared variable inside its constructor, if it is needed for lower level FSMs. Figure 6.6 shows the parameters for the constructor of FSM4.

```java
FSM4 (BaseFSM upper,
     String nameStr,
     InputSignal start_roc_in,
     OutputSignal rd_out,
     SharedVariable<Integer> roc_stat_in,
     SharedVariable<Float> ave_freq_in,
     SharedVariable<Float> roc_thresh1_in,
     SharedVariable<Float> roc_thresh2_in,
     SharedVariable<Float> roc_thresh3_in)

super(upper, nameStr, 3, rd_out);
```

Figure 6.6: Parameters of FSM4 constructor
6.1.4 Signal and shared variable connections, initialization of local variables

References to signals and shared variables that are passed to a constructor need to be assigned to variables in an FSM class. Figure 6.7 shows how this is done for FSM4. The initialization of local variables can also be seen in the Figure.

```cpp
start_roc = start_roc_in;
rd = rd_out;
roc_stat = roc_stat_in;
ave_freq = ave_freq_in;
roc_thresh1 = roc_thresh1_in;
roc_thresh2 = roc_thresh2_in;
roc_thresh3 = roc_thresh3_in;
first = 50;
second = 50;
roc_ws = 4;
roc_window = new float[roc_ws];
for(int i = 0;i < roc_ws;i++)
    roc_window[i] = 0;
```

Figure 6.7: Connections and initializations in FSM4

6.1.5 Linking states, transition inputs and transition outputs

In this part of the constructor, objects for states, transition inputs and transition outputs are created and then linked. Figure 6.8 show this for FSM4. The parameters of SimpleState constructor are the reference to the FSM the state belongs to, the number of outgoing transitions and the state name. HierarchicalState constructor has the same parameters. States, transition inputs and transition outputs are linked to make complete transitions by the method inherited from BaseFSM called connect(). The parameters of this method are source state, transition input, transition output, sink state and transition priority. The last statement in Figure 6.8 sets S41 as the initial state.
s1 = new SimpleState(this, 1, "s1");
s2 = new SimpleState(this, 1, "s2");
s3 = new SimpleState(this, 1, "s3");
s1input1 s1t1 = new s1input1();
s2input1 s2t1 = new s2input1();
s1output1 s1o1 = new s1output1();
s2output1 s2o1 = new s2output1();
s3output1 s3o1 = new s3output1();
connect(s1, s1t1, s1o1, s2, 1);
connect(s2, s2t1, s2o1, s3, 1);
connect(s3, s2t1, s3o1, s1, 1);
s1.setInitial();

Figure 6.8: Creating transitions in FSM4

The remaining sections of the constructor are only present in FSMs that are refined by other FSMs or SDFGs. Thus, we have to depart from FSM4. We will use FSM1 from the frequency relay specification for illustrations.

### 6.1.6 Local signals, shared variables and channels for lower level FSMs and SDFGs

Before lower level FSMs and SDFGs can be instantiated, the objects that enable communication between them (local signals, shared variables and internal channels) have to be created. Showing all local signals, shared variables and channels that are created within FSM1 would take too much space. Thus, we will only show in Figure 6.9 local signal `start_roc` that is used by FSM3 and FSM4, shared variable `ave_freq` that is also used by FSM3 and FSM4, and internal channel `ch2` that connects FSM3 and SDF1.

#### Local signal

A local signal is created as an instance of class LocalSignal. As we mentioned before, lower level FSMs are never aware that they communicate through a local signal. They see it either as an input signal or output signal. For that reason, besides an instance of LocalSignal, instances of InputSignal and OutputSignal also need to be created and passed to appropriate FSMs. Then they are linked with the instance of LocalSignal. For that purpose, both InputSignal and OutputSignal contain method setLocal() which
accepts the reference to a local signal. In Figure 6.9, start_roc_out and start_roc_in are passed to FSM3 and FSM4 respectively, and linked with start_roc_local.

Only a string denoting the signal name, has to be passed to the constructors of InputSignal and OutputSignal. The constructor of LocalSignal also requires the reference to the object that instantiates it, which can be either an FSM or top level.

In many applications, local signals are used for communication between FSMs that are on different hierarchical levels. For the higher level FSM, a local signal is always seen as an input signal. The reference for the InputSignal object has to be declared outside the constructor. Suppose that in the frequency relay specification, FSM1 also has to read start_roc besides FSM4. This situation would be handled in exactly the same way as in Figure 6.9, except that start_roc_in would be declared outside the constructor in the area where all input and output signals for FSM1 are declared. In this way, start_roc_in is visible to FSM1 transition inputs.

InputSignal start_roc_in = new InputSignal("start_roc_in");
OutputSignal start_roc_out = new OutputSignal("start_roc_out");
LocalSignal start_roc_local = new LocalSignal("start_roc_local",this);
start_roc_in.setLocal(start_roc_local);
start_roc_out.setLocal(start_roc_local);

SharedVariable<Float> ave_freq = new SharedVariable<Float>("ave_freq",0.0f,s12);
SDFFSMIntegerChannel ch2 = new SDFFSMIntegerChannel ("ch2");

Figure 6.9: Signal start_roc, shared variable ave_freq and channel ch1 in FSM1

**Shared variable**

Creating a shared variable is easier. There are three parameters that have to be passed to the constructor of SharedVariable: the name of the shared variable, the initial value, and the hierarchical state in which the shared variable is active. This is necessary to indicate when the shared variable has to be reinitialized. If a shared variable is created at the top level then it is always active. Instead of the reference to a hierarchical state the third parameter of the constructor would be the reference to the top level class.
It can be seen in Figure 6.9 that the initial value of $ave\_freq$ (average frequency) is 0 and it is active is state S12. Whenever FSM1 enters S12 $ave\_freq$ is initialized to 0. If FSM1 also needed to access $ave\_freq$, this shared variable would not be created in the constructor of FSM1. It would have to be created at the level above FSM1, which is the top level in this case.

Channel

When an FSM class obtains the reference to a shared variable, it is allowed to read and write the shared variable. It is up to the designer to ensure that in any tick only one FSM can write. If multiple updates occur, the result becomes unpredictable.

Currently, the DFCharts library supports passing arrays of two data types, integer and double, between FSMs and SDFGs in both directions. Thus, there are four different types of channels: FSMSDFIntegerChannel, SDFFSMIntegerChannel, FSMSDFDoubleChannel, SDFFSMDoubleChannel. The constructors of all four classes take only the channel’s name. All four classes have receive() and send() methods whose parameter is either integer or double array. In Figure 6.9, an SDFFSMIntegerChannel object is instantiated to enable communication between FSM3 and SDF1. It appears in the input of the transition that leads from S31 to S32 in FSM3. The inner class that specifies the transition input is shown in Figure 6.10.

```java
class s1input1 extends TransitionInput {
    public boolean evaluateInput() {
        if (ch1.receive(din))
            return true;
        else
            return false;
    }
}
```

Figure 6.10: The input for transition S31 → S32 in FSM3

Variable $din$ is a reference to an integer array. Its single element contains the number of samples between two consecutive peaks in the AC waveform. Receive() and send() methods return either true or false depending on whether the rendezvous is happening.
In Figure 6.10, if the rendezvous is happening on ch1 in the current tick, receive() puts data in din returns true. If not, receive() returns false while din remains unchanged.

### 6.1.7 Instantiation of lower level FSMs and SDFGs

When FSM and SDFG objects are instantiated all required parameters have to be passed to the constructors in the correct order. In HDL terminology, this would be positional association. Named association is not available. Figures 6.11, 6.12 and 6.13 show the instantiation of FSM3, FSM4 and SDF1 in the constructor of FSM1. The parameters of SDFG constructors will be described in section 6.2. After an SDFG has been instantiated, its channels have to be linked with it using method setGraph() as shown in Figure 6.13.

```java
FSM3 fsm3 = new FSM3 (this, "fsm3", rd_in, start_roc_out, freq_stat, ave_freq, freq_thresh1, freq_thresh2, freq_thresh3, ch2);
```

Figure 6.11: Instantiation of FSM3

```java
FSM4 fsm4 = new FSM4 (this, "fsm4", start_roc_in, rd_out, roc_stat, ave_freq, roc_thresh1, roc_thresh2, roc_thresh3);
```

Figure 6.12: Instantiation of FSM4

```java
int[] k = {5};
SDFPeriodCalculation sdfpc = new SDFPeriodCalculation (this, "sdfpc", 1, ch2,k);
ch1.setGraph(sdfpc);
```

Figure 6.13: Instantiation of SDF1

### 6.1.8 State refinement

After FSMs and SDFGs have been instantiated it has to be specified which states they refine. Those that refine the same state are executed concurrently. HierarchicalState class has method refine(), which is overloaded three times to support all three types of state refinement in DFCharts. It can accept an array of BaseFSM objects, an array of
BaseSDF objects, or both at the same time. Figure 6.14 shows the refinement of state S12 in FSM1.

```java
BaseFSM s1fsmr[] = {fsm2, fsm3, fsm4, fsm5, fsm6};
BaseSDF s1sdfr[] = {sdfpc};
s12.refine(s1fsmr,s1sdfr);
```

Figure 6.14: Refinement of S2 in FSM1

### 6.2 SDFG Classes

Each SDFG class has to extend the DFCharts library class called BaseSDF. The structure of an SDFG class is shown in Figure 6.15.

```java
public class SDF extends BaseSDF {

    SDF (internal channels, iteration length...) { // constructor
        base constructor parameters
        try {
            instantiation of SDF actors

            actor connections

        } catch {Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 6.15: Structure of SDF classes

As an example we use the code for SDF1 listed in Figure 6.16.
public class SDF1 extends BaseSDF {

SDF1 (BaseFSM container, String graphName,
    int channum, SDFFSMIntegerChannel ch1_in,
    int[] iterationLength) {
    super(container, graphName, channum, iterationLength);

    try {
        TestSamples ts = new TestSamples(top, "ts"); // sample source
        AverageFilter ave = new AverageFilter(top, "ave", 20); // averaging filter
        SymmetryDetection sym = new SymmetryDetection(top, "sym", 80); // symmetry detection
        PeakDetector pk = new PeakDetector(top, "pk"); // peak detector
        int[] intOutInit = {-1}; // initial token value
        OutputIntegerActor intOut = new OutputIntegerActor(top, "IntOut", 1, intOutInit); // interface actor
        ch1_in.setActor(intOut);

        top.connect(ts.output, ave.input);
        top.connect(ave.output, sym.input);
        top.connect(sym.output, pk.input);
        top.connect(pk.output, intOut.input);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

Figure 6.16: Class that specifies SDF1

6.2.1 Constructor parameters

The parameters of SDFG constructors are: the reference to the object that does the instantiation (FSM or top level), SDFG name, references to internal channels, and an integer array that indicates the length of SDF iterations in terms of FSM ticks. An array allows a repeating pattern of SDF iteration lengths. If all iterations have the same length then the array only needs to have a single element as in Figure 6.13. The parameters that have to be passed to the constructor of BaseSDF are: the reference to the object that does the instantiation, SDFG name, the number of internal channels and the iteration length array.

6.2.2 Instantiation of actors

Inside the try-catch statement, SDFG actors are instantiated and connected following the rules of Ptolemy. When actors are constructed, various exceptions that can be thrown
have to be caught. Actor constructors can have various parameters. A compulsory parameter that always has to be included is top, an instance of Ptolemy class TypedCompositeActor that is created in BaseSDF. An SDFG in a DFCharts specification is in fact seen as a TypedCompositeActor in Ptolemy.

Design of Ptolemy actors is described in [114]. We briefly illustrate it here using an example in Figure 6.17., which lists the code for the averaging filter in SDF1. AverageFilter extends Ptolemy class Transformer, which provides utilities for input-output operations. Variables input and output that appear in the constructor are inherited from this class. Their data type is set to DOUBLE, which is a replacement for the primitive type double in Ptolemy.

```java
public class AverageFilter extends Transformer {
    int AveIndex;
    int AveWs;
    double AveWindow[];

    public AverageFilter (CompositeEntity container, String name, int AveWsIn)
            throws NameDuplicationException, IllegalActionException {
        super(container, name);
        AveIndex = -1;
        AveWs = AveWsIn;
        AveWindow = new double[AveWs];
        input.setTypeEquals(BaseType.DOUBLE);
        output.setTypeEquals(BaseType.DOUBLE);
    }

    public void initialize() {
        for(int j = 0; j < AveWs; j++)
            AveWindow[j] = 0;
    }

    public void fire() throws IllegalActionException {
        double sample;
        sample = ((DoubleToken) input.get(0)).doubleValue();
        AveIndex = (AveIndex+1) % AveWs;
        AveWindow[AveIndex] = sample;
        double sum = 0;
        for (int j=0; j<AveWs; j++)
            sum = sum + AveWindow[(AveIndex+j) % AveWs];
        output.send(0, new DoubleToken(sum/AveWs));
    }
}
```

Figure 6.17: Averaging filter actor in SDF1
During an execution of a Ptolemy actor, methods preinitialize(), initialize(), prefire(), fire(), postfire() and wrapup() are invoked. These methods are not compulsory and most of actors in Ptolemy libraries do not have all of them. However, most have fire(), which is supposed to contain the main functionality of an actor. In AverageFilter, initialize() sets all samples in the buffer (AveWindow) to zero while fire() computes the average of all samples that are in the buffer.

SDFGs that communicate with FSMs, such as SDF1, have to include special interface actors. There are four interface actors corresponding to four types of channels: InputIntegerActor, InputDoubleActor, OutputIntegerActor and OutputDoubleActor. InputIntegerActor matches FSMSDFIntegerChannel, for example. Besides top, the constructor of an interface actor is passed the actor name, an integer showing the number of tokens communicated on the channel in a single rendezvous, and an array with initial token values if tokens flow from SDF to FSM. In Figure 6.17 the peak detection actor sends data to FSM3 by placing it in the interface actor intOut. When an interface actor is instantiated, the corresponding channel has to be linked with it using method setActor() as shown in Figure 6.16.

Sink and source actors that are needed for testing have to be attached to external channels. The simple input/output facility described in section 6.4 only handles FSM inputs and outputs. There are no sink and source actors in the DFCharts library since Ptolemy has plenty. It should be noted that the source actor used in SDF1 (class TestSamples) was not taken from a Ptolemy library.

6.2.3 Actor connections

The actors that make up a TypedCompositeActor are connected with its method connect(). While connecting outputs to inputs, it has to be ensured that data types are compatible [114].
6.3 Top level classes

A top level class is needed to instantiate and connect top level FSMs and SDFGs. It has to extend the DFCharts library class called DFChartsTop, which is in many ways similar to HierarchicalState. The structure of a top level class is shown in Figure 6.18.

```java
public class TopLevel extends DFChartsTop {
    TopLevel (input file, output file) { // constructor
        base constructor parameters

        instantiation of input signals
        instantiation of output signals

        instantiation of local signals for top level FSMs
        instantiation of shared variables for top level FSMs
        instantiation of channels for top level FSMs and SDFGs

        instantiation of top level FSMs
        instantiation of top level SDFGs

        top level refinement
    }
}
```

Figure 6.18: Structure of top level classes

The top level class for the frequency relay is shown in Figure 6.19.
6.3.1 Constructor parameters

The constructors of top level classes only have two parameters, the reference for the file that contains input stimulus, and the reference for the file where the outputs will be printed. The references for the two files are also passed to the base constructor.

```java
public class FrequencyRelayTop extends DFChartsTop {
    FrequencyRelayTop (String infName, String outfName) {
        super(infName, outfName);

        InputSignal on = new InputSignal("on", this);
        InputSignal off = new InputSignal("off", this);
        InputSignal reset = new InputSignal("reset", this);
        InputSignal sth = new InputSignal("sth", this);
        InputSignal cancel = new InputSignal("cancel", this);
        InputSignal done = new InputSignal("done", this);
        InputSignal thresh0 = new InputSignal("thresh0", this);
        InputSignal thresh1 = new InputSignal("thresh1", this);
        InputSignal skip = new InputSignal("skip", this);
        OutputSignal nt = new OutputSignal("nt", this);
        OutputSignal inth = new OutputSignal("inth", this);
        OutputSignal sw1 = new OutputSignal("sw1", this);
        OutputSignal sw2 = new OutputSignal("sw2", this);
        OutputSignal sw3 = new OutputSignal("sw3", this);

        FSM1 fsm1 = new FSM1 (this, "fsm1",
            on, off, reset,
            sth, cancel, done, thresh0, thresh1, skip, nt, inth,
            sw1, sw2, sw3);
        
        BaseFSM tr[] = {fsm1};
        topLevel(tr);
    }
}
```

Figure 6.19: Top level of frequency relay

6.3.2 Instantiation of input and output signals

All input and output signals, except those that serve as local signals, have to be instantiated at the top level and passed to FSMs. The constructors of InputSignal and OutputSignal only need the signal name and the reference to the top level class.
6.3.3 Local signals, shared variables and channels for top level FSMs and SDFGs

This section is the same as in FSM classes. The top level class of the frequency relay does not have it since there is a single FSM at the top.

6.3.4 Instantiation of top level FSMs and SDFGs

This section is also the same as in FSM classes. In the frequency relay, FSM1 accepts the references for all input and output signals even though it only uses on, off and reset. The rest is passed to lower level FSMs.

6.3.5 Top level refinement

The top level can be seen as always active hierarchical state that does not belong to any FSM. Hence, it is “refined”. The topLevel() method in DFChartsTop is very similar to refine() in Hierarchical state. It is overloaded three times to support the three refinement types. FSM and SDF objects have to be put into arrays before they are passed to topLevel().

6.4 Simulation

Finally, we need a class that runs the top level class. The top level class is executed with runTopLevel(). The class that runs the frequency relay top level is shown in Figure 6.20.

```java
public class FrequencyRelay {
    public static void main (String args[]) {
        FrequencyRelayTop relay = new FrequencyRelayTop("frequency_relay_input.txt","frequency_relay_output.txt");
        relay.runTopLevel();
    }
}
```

Figure 6.20: Execution of the top level class of the frequency relay
The input and output files passed to the top level class only contain FSM inputs signals and outputs signals. As mentioned before, SDF graphs have separate source and sink actors for testing.

In the input file, the status of each input signal has to be defined in each tick. The name of an input signal and its status (present or absent) are written on separate lines. Ticks are separated by blank lines. If the same inputs repeat over successive ticks, the instruction ‘repeat’ can be used with the number of ticks that have identical inputs. The simulation is terminated by writing ‘end’. An input file for a system with two inputs could look as in Figure 6.21

```
input1
present
input 2
absent
repeat
10
end
```

Figure 6.21: Input file format

In the output file, outputs are printed in the format that is used for inputs. In addition, the current state of each FSM is printed in every tick.

### 6.5 Library classes

DFCharts library classes enable execution of FSM and SDF classes described in the previous sections by providing synchronization and communication mechanisms. It should be noted that the library does not yet include any classes that identify causality cycles in DFCharts specifications. Thus, it is currently the responsibility of a designer to ensure that communication between FSMs is valid. If FSMs are incorrectly connected with instantaneous loops, a simulation will deadlock. On the other hand, if any of SDF graphs in a specification is not constructed correctly, Ptolemy software will throw an exception.
In total there are 23 classes in DFCharts library. They can be divided in five groups: base classes, FSM component classes, FSM communication classes, FSM – SDF communication classes, and synchronization class. The base classes include BaseFSM, BaseSDF, DFChartsTop; the FSM component classes include SimpleState, HierarchicalState, TransitionInput, TransitionOutput, Transition; FSM communication classes include InputSignal, OutputSignal, LocalSignal, SharedVariable; FSM–SDF communication classes include FSMSDFChannel, SDFFSMChannel, FSMSDFIntegerChannel, SDFFSMIntegerChannel, FSMSDFDoubleChannel, SDFFSMDoubleChannel, InputIntegerActor, OutputIntegerActor, InputDoubleActor, OutputDoubleActor. The single synchronization class is ThreadControl.

Most of these classes have already been described in the previous sections to some extent. In the following sections further details will be added for each group. In addition, relations among different groups will be highlighted.

### 6.5.1 Base classes

FSM and SDF objects, which inherit from BaseFSM and BaseSDF, need to be able to run as separate threads. For this reason, both BaseFSM and BaseSDF implement the Runnable interface. A DFCharts top object is attached to the main thread. At any point in the execution of a DFCharts specification in Java, the number of active threads is equal to the sum of all active non-refined FSMs and all active SDFGs plus the main thread. Active FSMs that are refined do not consume additional threads. The thread of a refined active FSM simply takes over one of the refining FSMs.

BaseFSM provides connect() method for construction of an FSM, which links states with transitions. It also has variables for current state and next state, which are necessary for execution of an FSM.

The main purpose of BaseSDF is to provide an interface with Ptolemy software, which is necessary for execution of SDF graphs. Therefore, when DFCharts library is built, Ptolemy must be included in the build path. From the Ptolemy’s perspective, each SDF graph is an instance of TypedCompositeActor, which is called top in the DFCharts
library. The execution of a TypedCompositeActor is handled by methods provided by class Manager. Those methods are invoked from the run() method of BaseSDF. When an SDFG is started, initialize() is invoked. For each iteration, iterate() is invoked. Finally, when an SDFG is stopped, wrapup() is invoked.

A DFChartsTop object spawns the threads for top level FSMs and SDFGs when runTopLevel() calls one of the three methods - runType1(), runType2() or runType3(), depending on the contents of the top level.

### 6.5.2 FSM component classes

All FSM component classes except Transition represent inputs for the connect() method of BaseFSM. The Transition class is used in the data structure that is built inside BaseFSM. It contains reference variables for TransitionInput and TransitionOutput.

The most important function of SimpleState is to determine which transition should be taken and produce the outputs for the selected transition. For that purpose, the method evaluateTransitions() is used. It evaluates transitions inputs in the order of their priorities. When it finds a transition that is enabled, it produces the outputs for that transition. The next state cannot be immediately set since it may be possible that the FSM is pre-empted in the current tick. Consequently, the evaluateTransitions() method of the refined state is called. This is a recursive process that leads to a top level FSM. When it is finished, the next state variable in all surviving FSMs can be updated.

HierarchicalState also has to evaluate transitions but it has an important additional task. It has to spawn threads for the refining FSMs and SDFs. In this respect, it is very similar to DFChartsTop. If a state is refined only by FSMs, the method runType1() is used. If it is refined by an SDGF, runType2() is used. It should be noted that a state can also be refined by several disconnected, independently operating SDFGs, but this is not a usual situation. If a state is refined by both FSMs and SDFGs, runType3() is used.
TransitionInput and TransitionOutput are abstract classes with no functionality, which contain abstract methods evaluateInput() and computeOutput() respectively. Their purpose is to facilitate building the data structure that represents an FSM.

6.5.3 FSM communication classes

InputSignal and OutputSignal enable communication between FSMs and the external environment. Both are relatively simple as they only contain two methods, read() and write(). If the status of a signal is present boolean true is written, otherwise false is written.

LocalSignal is more complicated as it has three values: present, absent and unresolved. When a new tick begins, the value of a local signal remains unresolved until an FSM writes true or false. If another FSM attempts to read the local signal while it is still unresolved, its thread will become blocked. It will be notified when the value of the signal becomes true or false.

SharedVariable is simpler than LocalSignal but more complicated than InputSignal and OutputSignal. It has write() and read() methods which work with all primitive data types. It also contains a flag that indicates when a shared variable is active. A shared variable is reset to its initial value only in ticks in which it gets activated. By comparison, LocalSignal does not need a similar flag since it does not have memory. At the beginning of each tick, all local signals are set to unresolved.

6.5.4 FSM-SDF communication classes

In this group FSMDFChannel and SDFSMChannel are base classes. FSMDFIntegerChannel and FSMDFDoubleChannel extend FSMDFChannel, while SDFSMIntegerChannel and SDFSMDoubleChannel extend SDFSMChannel. Channel classes enable communication between FSMs and SDFGs in both directions using arrays of two primitive data types, floats and integers. Arrays always have to be used. If there is only a single value flowing through a channel, an array of size one is used. Communication is performed by receive() and send() methods. The parameter of
both methods is the reference to an array, which is read from or written to depending on which method is used. Both methods return true if rendezvous occurs, or false otherwise. Channel classes also contain control variables which ensure that data flows through a channel only once in a single rendezvous between an FSM and an SDFG.

Input interface actors are essentially source actors in an SDGF when communication is performed with an FSM. They are included in the iteration schedule of an SDFG, just like other actors. The send() method of the channel classes places tokens (floats or integers contained in an array) into an input interface actor. During an iteration of an SDFG, these tokens are read out. It has to be ensured that as many tokens are placed as needed. Otherwise, the input interface actor will output some values multiple times, which would most likely lead to incorrect behaviour.

Similarly, output interface actors can be thought of as sink actors in an SDGF. During an iteration, tokens are placed inside an output interface actor. They are brought to an FSM by the receive() method of the channel classes. As in the case of input interface actors, it needs to be ensured that the right number of tokens is placed in output interface actors.

### 6.5.5 Synchronization class

The single class that falls under this category is called ThreadControl. Synchronizing threads is its main purpose, but it additionally performs several other actions that occur at the end of each tick. Among those actions is ensuring that each SDF iteration lasts the specified number of ticks.

When an FSM completes a transition, its thread blocks as it encounters wait(). When all FSM threads block, the tick is completed. At this point, methods in ThreadControl print output signals, read inputs for the next tick, clear local signals and update shared variables. Then, if there are any SDFGs that are due to complete an iteration, ThreadControl waits before starting a new tick. A new tick is started by unblocking all threads using notify().
6.6 Frequency relay in SystemC, Esterel and Java

When compared against its counterparts in SystemC and Esterel, the most important feature of the frequency relay specification in Java is that it fully conforms to the DFCharts model from section 3.2. The specifications in SystemC and Esterel contain alternative solutions for the parts of the DFCharts model that the two languages could not completely support.

The size of the Java specification is 1479 lines. This appears to be significantly longer than 1102 lines of SystemC code and 901 lines of Esterel code. However, the difference is entirely due to numerous class declarations that have to be made in Java. As we explained in section 6.1 inputs and outputs of transitions have to be specified as classes. What really matters is that describing the actual behaviour does not take more effort in Java than in SystemC and Esterel. Class declarations can be easily handled with templates. Hence, they should have no impact on design time.
Chapter 7

Implementation of DFCharts on HiDRA

Previous chapters deal with system level by exploring issues such as specification, simulation and formal semantics of DFCharts without any reference to implementation. The focus of this chapter is implementation of DFCharts specifications. The target architecture for implementation is HiDRA, which was described in the background chapter. HiDRA is capable of implementing both control-dominated and data-dominated types of behaviour that are found in DFCharts. It has special features supporting reactivity while data-dominated operations are supported using traditional solutions. In principle, any multiprocessor architecture may be used for implementation of DFCharts. However, because of special features that support reactivity, HiDRA is likely to provide more efficient implementations. This is the main reason for selecting HiDRA. In section 7.1, we present a design methodology for implementing DFCharts on HiDRA. It consists of five steps: specification, FSM compositions, allocation and partitioning, synthesis and performance evaluation. We are mainly concerned with the synthesis step, which is separately treated in section 7.2 where details regarding the execution of DFCharts on HiDRA are presented. In section 7.3 we show how the methodology is applied on the frequency relay case study.
7.1 DFCharts design methodology

The design methodology is presented by the design flow that consists of five steps shown in Figure 7.1. The first step is to make the DFCharts-based specification of a system without any reference to HW/SW implementation. Before the specification is mapped onto the implementation architecture, the designer has an opportunity to merge multiple FSMs into a single equivalent FSM by parallel and hierarchical compositions.

![DFCharts based design flow](image-url)

Figure 7.1: DFCharts based design flow
In the third step, an instance of HiDRA is created by connecting ReMICs and functional units. Then, FSMs and SDFGs are partitioned among the processing elements. As this step is difficult to automate, the designer can freely explore different implementation options. A complete implementation is created by automatically synthesizing FSMs and SDFGs into ReMIC instructions and RTL in the fourth step. Its performance is evaluated in the fifth step. If the implementation is not satisfactory, the designer can return to any of the first three steps.

A key feature of the methodology is that functionality and implementation are completely separated. Functionality is only verified at the specification level, using simulation and formal verification.

### 7.1.1 Specification

Specifications are created in Java environment described in Chapter 6. Currently, textual design entry is only supported, but a graphical user interface can be added later, which would enable specifications to appear as in Chapter 3. It is also possible to employ SystemC or Esterel for creating specifications after modifying them according to the guidelines given in Chapter 5. It is important to note that when an SDFG is created its relative speed is specified by indicating how many ticks of the FSM clock each iteration takes, as pointed out in section 6.1.2.

As indicated in Figure 7.1, validation can be done using simulation. Simulation of DFCharts-based designs in Java has been described in detail in Chapter 6. While simulation is the most widely used means of validation, formal verification has been gaining importance. Formal verification of DFCharts has not been discussed in this thesis. An efficient method for formal verification of DFCharts can be developed using the MCFSM model. This will be an important future research direction.

### 7.1.2 FSM compositions

There is overhead associated with running an FSM on a processor or functional unit. For example, variables must be used to remember the current state of an FSM and
indicate if it has made a transition in the current tick. Local signals that are used for communication between FSMs always consume resources even if the FSMs are mapped onto a single processing element. When an FSM composition is applied on two FSMs, they will be executed as a single equivalent FSM. Two important effects are produced as a result. The overhead is reduced, since there is now one FSM instead of two, and local signals are removed. Thus, FSM compositions give the designer an opportunity to trade concurrency for more efficient execution. Two types of FSM compositions are used: parallel and hierarchical. The two compositions and their relation to the synchronous parallel and hierarchical operators will be discussed in more detail in section 7.2.3. At this point it is important to stress that the parallel composition produces an exponential increase in the number of states, while the hierarchical composition produces a linear increase in the number of states. For example, looking at the frequency relay specification, if FSM5 (4 states) and FSM6 (2 states) from Figure 3.8 are merged by the parallel composition, the resulting FSM has 8 states. If FSM2 (3 states) from Figure 3.9 is merged with FSM7 (7 states) from Figure 3.10 by the hierarchical composition, the number of states in the resulting FSM increases linearly to 9.

7.1.3 Allocation and partitioning

An instance of HiDRA is created by connecting ReMICs and hardware-implemented functional units. Communication between processing elements is performed through shared memories and ReMIC signal lines. How many processing elements are allocated depends on the desired performance. A higher number of elements may provide better performance, but the cost increases at the same time.

An SDFG can be mapped on a ReMIC or functional unit. Digital signal processors (DSP) or application specific processors could also be included in the architecture for SDFG implementation as long as they satisfy some simple interface requirements. Multiprocessor implementation of a single SDFG is possible [23]. Even mixed HW/SW implementation for SDF has been demonstrated [53]. We could easily incorporate those techniques in our methodology. However, we will assume that each SDFG is executed by a single processing element in order to simplify the discussion. FSM can also be
mapped on a ReMIC or functional unit. Although, performance boost provided by hardware is more likely to be needed for computationally intensive SDFGs. FSMs and SDFGs cannot be mapped on the same element. The reason for this restriction is due to different implementation requirements posed by FSMs and SDFGs. FSMs react to events and perform minor computations. Reactivity is not important for SDFGs but computations are intensive. By executing only one type of behaviour, processing elements can be customized to do their tasks efficiently with minimum resources.

ReMIC consists of three parts: reactive functional unit (RFU), control unit and datapath. If it executes an SDFG, RFU becomes unnecessary. When RFU is removed, ReMIC becomes a Harvard-type microprocessor with RISC-type pipelined datapath, suitable for transformational operations performed by SDFGs. On the other hand, an FSM needs RFU but the datapath does not need to be as powerful. ReMIC has a customizable register file with up to 16 registers. If it implements an SDFG it may need all 16 registers. In the case of FSM implementation, just two might be enough. Furthermore, pipelining, which is useful for predictable SDF data transformations, could be redundant for FSMs.

When FSMs are partitioned among multiple processors, the key consideration is how to maximize the utilization of the processors. We say that a global tick is completed when each FSM in the system makes a transition (completes its local tick) by computing outputs and setting the next state. Because of the lock-step execution assumed at the specification level, it must not happen that an FSM executes two transitions in a sequence while another one executes none. When all FSMs on a single processor have finished their local ticks, the processor tick has been completed. At this point the processor suspends execution and waits for the remaining processors to complete their ticks, so that a new global tick can begin. In the ideal case, all processor ticks take equal time which means that all processors are fully utilized.

Two factors are important for maximizing processor utilization: distribution of loads and inter-processor communication. By loads, we mean execution times of FSM transitions. Loads should be distributed as evenly as possible. In any state of the system, the sum of execution times of FSM transitions should not differ largely across the
processors. This may be a difficult task considering that the system may have a large state space resulting in many combinations that have to be explored. However, the designer can be assisted by profiling tools that indicate the amount of time consumed by various execution paths. At this stage, before the synthesis has been completed, only approximate times can be provided. Cycle-accurate performance results are examined in the fifth step.

Balancing of loads is additionally complicated by communication dependencies between FSMs that are executed by different processors. It may happen that a transition of an FSM cannot be completed because it depends on a transition of another FSM that is executed on another processor. Therefore, it is desirable to locate FSMs that communicate by signals on a single processor.

Among the processors executing FSMs, one must be designated as master processor. The others are called slave processors. Apart from running its FSMs, the master processor has the additional tasks of transferring data across rendezvous channels, controlling the execution of SDFGs, and managing global ticks.

Mapping SDFGs takes less effort. SDF inputs usually arrive in regular intervals from the external environment. The main issue is whether a processor executing an SDFG is fast enough to complete an iteration before next inputs arrive. It is possible to map multiple SDFGs on a single processor, if it is fast enough to service all of them.

7.1.4 Synthesis

A complete implementation is obtained by synthesizing ReMIC instructions, RTL for functional units, and communication between processors which is realized by ReMIC signal lines and shared memories. The program memory of a ReMIC executing FSMs consists of three sections: FSM threads (FT), FSM scheduler and tick handler. If SDFGs are executed, the program memory consists of two sections: SDF threads (ST) and SDF scheduler. An FSM thread implements the functionality specified by an FSM. An SDF thread implements the functionality specified by an SDFG. It simply consists of code that implements SDF actors. SDF scheduler, FSM scheduler and tick handler can be
considered as ‘middleware’ that enables threads to run. An SDF scheduler invokes SDF actors according to a static schedule. It also has to implement a simple interface with the master processor. In section 7.2 we will not deal with the SDF scheduler since it is described in detail in [23] and many other publications that followed. An FSM scheduler runs FTs in a round robin fashion according to a statically determined schedule. It is possible that an FSM thread is not ready to make a transition when picked by the scheduler due to unresolved local signals. When the scheduler encounters such FSM thread, it selects the next FT, but it will come back in the next cycle. A signal is unresolved if the thread that writes it has not decided yet whether it is present or absent in the current tick. Obviously, a thread attempting to read an unresolved signal cannot proceed. The scheduler has to repeat the schedule until all FSM threads have made a transition. The control then goes to the tick handler. Figure 7.2 shows the general flow of control between FTs, FSM scheduler and tick handler. In hardware-implemented functional units, concurrent executions are possible without the need for schedulers.

Figure 7.2: Flow of control between FTs, FSM scheduler and tick handler

The scheduler and FSM threads appear exactly the same on both the master and slave processors. However, tick handlers are different. When a slave tick handler gets control, it informs the master processor that all of its FSM threads have completed a transition. When the master tick handler receives this message from all slave processors the global tick is completed. The master tick handler instructs all slave tick handlers to update shared variables and clear written signals before the next global tick begins. It also does that but has to perform a few additional actions.
7.1.5 Performance evaluation

Although estimations in step 3 could provide useful feedback about a particular implementation, cycle accurate results obtained after synthesis are needed to show precisely whether various constraints are satisfied. A typical requirement in a DFCharts-based specification is to ensure that an SDFG does not miss any input samples from the external environment. According to the DFCharts semantics, the communication on external SDF channels is performed by rendezvous as on the internal channels. However, due to the nature of embedded systems, the external environment will not really wait. If an SDFG is not ready to receive a sample when it arrives, the sample will be inevitably lost unless it is buffered. For many applications though, buffering on input channels is not a satisfactory solution since it may result in unacceptable delays.

7.2 Execution of DFCharts specifications on HiDRA

7.2.1 Signals and variables

Signals and variables enable synchronization and communication among code sections. The most important difference between signals and variables is in their physical implementation. Variables are always implemented with memory. Signals can be implemented with memory or ReMIC signal lines that are linked with special instructions supporting reactivity.

Signals can either be visible in the specification or inserted during synthesis. The former will be called specification-visible signals while the latter will be called specification-invisible signals. The specification-visible signals can be input, output or local. Furthermore, we distinguish between external and internal signals. An external signal is used for communication between a processing element and the external environment. An internal signal is used for communication between two processing elements. External signals include specification-visible input and output signals. Internal signals include specification-visible local signals and specification-invisible signals. Specification-invisible signals are listed below. Some signals that are involved in the
same type of operation are grouped together and only briefly described. Their use will become obvious in later sections.

1. FT_start: Written by an FT upon entering a hierarchical state. Read by FTs that refine the hierarchical state. May be written and read any time during the tick execution.
2. FT_stop: Written by an FT upon leaving a hierarchical state. Read by FTs that refine the hierarchical state. May be written and read any time during the tick execution.
3. ST_start_request: Written by an FT upon entering a hierarchical state any time during the tick execution. Read by the master processor’s tick handler at the end of the global tick.
4. ST_stop_request: Written by an FT upon exiting a hierarchical state any time during the tick execution. Read by the master processor’s tick handler at the end of the global tick.
5. ST_start: Written by the master processor’s tick handler at the end of the global tick. Read by an ST. Unlike the signals described above, this type of signal is held high briefly, just long enough for the ST to respond.
6. ST_stop: Written by the master processor’s tick handler at the end of the global tick. Read by an ST. Like the ST_start signals, this type of signal is a short pulse.
7. FT_rendezvous_ready: Written by an FT upon entering a rendezvous state any time during the tick execution. Read by the master processor’s tick handler at the end of the global tick.
8. ST_rendezvous_ready: Written by an ST at the end of an iteration. Read by the master processor’s tick handler at the end of the global tick.
9. Rendezvous_done: Written by the master processor’s tick handler at the end of the global tick when a data transfer between shared memories has been done. Read by an FT any time during the tick execution and by an ST between two iterations.
10. Tick_finished, Start_tick_end_actions, Tick_end_actions_completed, Start_tick: Read and written by tick handlers at the end of the global tick. The purpose of
these signals is to synchronize tick handlers so that certain actions are done in the right order before the next global tick begins.

11. Update_shared_variable: Written by the thread that writes the shared variable. Read by the master processor’s tick handler at the end of the global tick.

Two bits are needed to implement every internal signal that can be written and read any time during the tick execution. Signals that need two bits are specification-visible local signals. Among specification-invisible signals, FT_start and FT_stop signals need two bits. In each case, one bit is needed to indicate whether the signal has been resolved while the other one shows the status of the signal. If writing and reading threads are on the same processor the bits are contained inside the local memory. If they are distributed on multiple processors then the bits must be implemented by input/output registers. Both solutions may be needed depending on how threads are allocated. Currently memory mapped registers sir and sor are used. With minor modifications of ReMIC, the registers sip and sop that implement reactive signals can be employed. The modifications would ensure that an emitted signal is held high for the duration of the whole tick instead of just one processor cycle. In the current form, reactive signals can be used just for external input and output signals.

LDR R0 m
LDR R1 #b
CLFZ -- clear zero flag
AND R2 R0 R1
SZ -- skip if zero PRESENT r

(a) testing a memory bit (b) testing a reactive signal

Figure 7.3: Comparing memory and reactive signals

If ReMIC’s reactive signals were used for implementation of internal signals, the code size would be reduced significantly since fewer statements are needed for reading and writing when using emit and present instead of ordinary instructions. The difference between using reactive signals and memory is shown in Figure 7.3. Part (a) shows that
five instructions are needed for testing a bit in memory \( m \). The test is performed by ANDing the memory contents loaded in register R0 with the bit pattern \( b \) loaded in register R1. If the result is zero then the bit under test is zero. Part (b) shows that only one instruction is needed for testing reactive signal \( r \).

We can divide variables into specification-visible and specification-invisible, as we did with signals. A specification-visible variable can be a shared variable that is used by multiple FSMs or local variable that is used by a single FSM. The main specification invisible variables are listed below:

1. \textit{next\_thread}: It points to a code section in the FSM scheduler. It indicates which FT the FSM scheduler will attempt to run next.
2. \textit{next\_state}: Each FT has this variable. It indicates which state the FT will take in the next tick.
3. \textit{pe}: (program counter). Each FT has this variable. It points to a section of code in an FT.
4. \textit{threads\_done}: Each bit in this variable is used to indicate whether an FT has completed its local tick in the current global tick.

For each specification-visible shared variable, two locations in memory are needed. One location is needed for the current value of the shared variable while the other is needed for storing the value that the shared variable will take in the next tick. If the writing thread and the reading threads are all executed on a single processor, the locations needed for the shared variable will be in the processor’s local data memory. Otherwise the shared memory which is accessible by the master processor must be used. When transferring data from one shared memory to another, the master processor reads a value from one shared memory into a register, and then writes the value to the other shared memory using the same register.

The flow of data together with signals connections is shown in Figure 7.4 for a configuration of five ReMICs that implement six FSMs and two SDFGs. When large amounts of data are involved in transfers between shared memories, a DMA controller
may be a useful addition to the architecture, which would enable data to flow directly between shared memories instead of going through the master processor.

Before describing FSM thread and other code sections on a ReMIC executing FSMs, we explain the notation and terminology that we will use. In figures that present implementation templates, variable names are in italics. Signal names are not italicized but they always begin with an upper case letter. Constant always begin with an underscore as in _number_of_slave_processors. We use brackets when we want to relate a signal to a particular object like FT, state, channel etc. For example, FT_stop(ft1) means that an FT_stop signal is used by the FSM thread ft1. Labels for code sections are underlined. For internal signals that are implemented by two bits, we use “emit” and “resolve” to denote setting high the status and resolution bits respectively. For internal signals implemented with a single bit, we simply use “set high”. If a signal has been set high in the current tick, “cancel” can be used to set it low.

Figure 7.4: Architecture for DFCharts implementation

7.2.2 FSM thread

An FSM thread (FT) implements an FSM. It has a section of code for each FSM state visible in the specification, although the number of FSM states may be reduced by
optimization techniques such as bisimulation before implementation. Additionally, three sections of code appear: entry, exit and tick end. Only FTs that implement preemptable FSMs contain the entry and exit code. On the other hand, every thread must have the code for tick end. The template for an FT called \( ft \) implementing an FSM with \( m \) states is given in Figure 7.5.

\begin{verbatim}
ft_entry:  if FT_start(ft) is not resolved then 
      set pc to ft_entry 
      goto section pointed by next_thread
else if FT_start(ft) is present then 
      set pc to ft_s1_t1 (highest priority transition of initial state \( s_1 \)) 
      emit and resolve FT_start signals for FTs that refine \( s_1 \) 
      set high ST_start_request signals for STs that refine \( s_1 \) 
      if \( s_1 \) is a rendezvous state then 
        set high FT_rendezvous_ready(s1) 
      end if 
      goto ft_tick_end
else 
      set pc to ft_entry 
      goto ft_tick_end
end if 

ft_s1_t1:  if not all signals in the trigger are resolved then 
      set pc to th1_s1_t1 
      goto section pointed by next_thread
else if trigger and variable conditions are true then 
      emit and resolve specification-visible local signals 
      set high specification-visible output signals 
      do procedures 
      determine next state and write it into next_state 
      emit FSM_start signals for FTs that refine next state 
      emit and resolve FSM_stop signals for FTs that refine this state 
      set high ST_start_request signals for STs that refine next state 
      set high ST_stop_request signals for STs that refine this state
\end{verbatim}
if this state is a rendezvous state then
    set low FT_rendezvous_ready signal for this state
end if
if next state is a rendezvous state then
    set high FT_rendezvous_ready signal for next state
end if
goto ft exit
end if

ft s1 t_11

ft s2 t_21

ft s_m t_m

ft exit: if FT_stop(ft) is not resolved then
    set pc to ft exit
goto section pointed by next_thread
else if FT_stop(ft) is present then
    emit and resolve FSM_stop signals for lower level FTs
cancel FSM_start signals for lower level FTs
    set high ST_stop_request signals for lower level STs
cancel ST_start_request signals for lower level STs
goto ft entry
else
    copy next_state to th1_pc
goto ft tick end
end if
ft tick end: resolve all unresolved signals
    set high the ft bit in threads_done
goto tick handler

Figure 7.5: FSM thread
Thread entry

A preemptable FSM thread ft is initially in ft_entry. The signal FT_start(ft) determines whether ft will start by entering the initial state s₁. FT_start(ft) is written by the FT which has a state refined by ft. If FT_start(ft) is not resolved the control will jump to a location in the scheduler which is pointed by the next_thread variable. pc will still be set to ft_entry. However, ft has not finished its local tick which means that the scheduler will have to return to ft before the next global tick can begin. If FT_start(ft) is present, the next state will be the initial state. If the initial state is hierarchical, FT_start signals are emitted and resolved for all FTs that refine the initial state, and ST_start_request signals are set high for all STs that refine the initial state. If FT_start(ft) is absent, ft will remain in ft_entry. When FT_start(ft) is resolved the local tick is completed after executing the necessary instructions, so the control jumps to ft_tick_end. It does not go to ft_exit since an FT cannot be entered and exited in the same tick. The other way around is possible though. An FT can be exited and entered in the same tick when the hierarchical state that is refined by it makes a transition back to itself.

It is important to notice here that ft has to complete its local tick even though it has not yet entered the initial state that is visible in the specification. From the point of view of the tick handler, each thread is always active. If a preemptable thread is waiting to enter the initial state it may be said that it is in the “pre-initial” state and hence it has a tick to complete. In this way there is no need for a data structure which has to distinguish between active and inactive threads.

States visible in specification

The code for each state in ft consists of one or more outgoing transitions. In Chapter 4, a state had to have an outgoing transition for every input combination for the purpose of analysing reactivity and determinism. Many of those transitions looped back to the source state without producing any outputs. Such transitions are not implemented. If a transition does not leave a state, it is implemented only if it produces outputs i.e. it emits signals or calls procedures. In Figure 7.5 state s₁ has n₁ transitions; state sₘ has nₘ transitions. Each transition is realized with a two branch if-else construct. The if-else
constructs are arranged in the order of transition priorities. The transition with the highest priority is listed first. In each transition, it is first checked whether the local signals in the transition trigger are resolved. A Rendezvous_done signal may also be a part of the trigger but, as for input signals, the resolution is not an issue. If not all signals are resolved, the next FT is selected by the scheduler but the current state of ft will have to be visited again since the local tick has not been completed. If all signals are resolved, the trigger and variable conditions are evaluated to determine whether the transition should be executed. If it is false the next transition is executed. Otherwise the transition outputs are produced. Specification-visible signals are emitted. Local signals have to be resolved in addition. It is possible to immediately resolve local signals due to the absence of strong abort in DFCharts. Procedures are also executed. If a shared variable is updated inside a procedure the corresponding Update_shared_variable signal must be set high. The next state is determined, but pc is not immediately set to the next state because it has to be checked in ft exit if ft has been pre-empted. If the next state is hierarchical, FT_start signals for FTs that refine it are emitted but they cannot be resolved immediately since they may be cancelled in ft1 exit if it turns out that ft has been pre-empted. FT_stop signals are emitted and resolved immediately for FTs that refine the current state. Start request signals are set high for STs that refine the next state while stop request signals are set high for STs that refine the current state. If the current state makes a transition back to itself, FT_stop and FT_start signals are emitted for the same FTs. The same applies for STs. The FT_rendezvous_ready signals are also handled if the current or next states are rendezvous. After all the outputs have been created the control jumps to ft exit.

Thread exit

In ft exit it is checked by reading signal FT_stop(ft) whether ft has been pre-empted by the higher level thread. If the preemption has taken place the FT_stop signals for lower level FSM threads are emitted and resolved. This is necessary in case the current state is hierarchical and no transitions have been taken out of it. The same is done for lower level STs. Furthermore all emitted FT_start signals and ST_start_request signals are cancelled. The control jumps to ft entry, not to ft tick end, since ft may be restarted in
the current tick. If the preemption has not taken place \( pc \) is simply set to the next state and \texttt{ft tick end} is executed next.

**Local tick end**

In \texttt{ft tick end}, \( ft \) acknowledges that it has completed its local tick by asserting its bit in \texttt{threads_done} variable. Also, all signals that have not been resolved previously are resolved here. This is necessary since only emitted signals are resolved in code sections for transitions, not ones that are absent.

### 7.2.3 Hierarchical and parallel compositions

Figure 7.6 shows a DFCharts specification consisting of three FSMs. Without any FSM compositions, three FTs are needed for implementation. Figure 7.7 shows the flow of control among FTs, scheduler and tick handler for a single processor implementation. \( ft1, ft2 \) and \( ft3 \) implement FSM1, FSM2 and FSM3 respectively. Since \( ft3 \) is a preemptable FSM thread, it has to have thread entry and thread exit sections where it reads \texttt{FT_start} and \texttt{FT_stop} signals that are written by \( ft1 \). In general, these signals can be unresolved when they are read. For this reason, when the control reaches \texttt{ft3 entry} or \texttt{ft3 exit} it may immediately flow back to the scheduler. Similarly, local signal \( b \) can be unresolved when read by \( ft1 \). Thus the control may flow from \texttt{ft1 s12 t1} to the scheduler. A schedule can take into account communication dependencies between FTs so that \texttt{FT_start}, \texttt{FT_stop} and \( b \) are always resolved when they are read. If \( ft1 \) is executed before \( ft3 \) in each global tick, \texttt{FT_start} and \texttt{FT_stop} will always be resolved when read. The same can be achieved for specification-visible signal \( b \) if \( ft2 \) is executed before \( ft1 \).

The hierarchical composition of two FSMs follows the semantics of the refinement operator from section 4.1.5. Figure 7.8 shows the implementation of the specification from Figure 7.6, where \( ft1 \) implements the hierarchical composition of FSM1 and FSM3 while \( ft2 \) implements FSM2 as in Figure 7.7. The FSM that represents the hierarchical composition of FSM1 and FSM3 has three states: \texttt{S11S31}, \texttt{S11S32} and \texttt{S12}. \texttt{FT_start} and \texttt{FT_stop} signals are no longer needed. Thread entry and thread exit...
section from Figure 7.7 also disappear. However, FSM1 and FSM3 cannot be executed concurrently by different processors. The potential benefit of executing proc1 and proc3 simultaneously would no longer be available.

The parallel composition of two FSMs follows the semantics of the synchronous parallel operator from section 4.1.2. The states of the resulting FSM are created by taking the cross product of the states that belong to the input FSM threads. As with the hierarchical composition, internal signals are removed, but concurrency disappears as well. The additional concern with the parallel combination is the potential state explosion that results from the cross product. For example, if FSM1 and FSM2 from Figure 7.6 are merged, local signal b is no longer needed and the resulting FSM has 6 states: S11S21, S11S22, S11S23, S12S21, S12S22 and S12S23.

![Diagram of three FSMs](image)

Figure 7.6: A DFCharts specification consisting of three FSMs
Figure 7.7: Implementation of specification from Figure 7.6 without any FSM compositions
In the DFCharts automata semantics the operators are applied to create a single FSM that represents the behaviour of a whole specification. The equivalent FSM is constructed bottom-up, by starting from the lowest hierarchical levels and moving upwards to the top level. For example, in Figure 7.6, FSM1 and FSM3 would have to be
merged first. The result is then combined with FSM2. This restriction does not apply here. FSM1 and FSM2 can be merged by the parallel composition while leaving FSM3. Before the parallel composition, FSM3 refines state S11. After the parallel composition, FSM3 refines states S11S21, S11S22 and S11S23.

7.2.4 FSM Scheduler

The template for the FSM scheduler is shown in Figure 7.9. It specifies the order in which threads are run. The schedule is static; threads are run in the same order in every processor tick. After the system’s start-up the processor begins execution at run_ft1. In each processor tick, the schedule is repeated until all threads have completed their local ticks. If a thread has already completed its local tick, it is skipped.

```plaintext
run_ft1:         if this thread completed its tick (its bit in threads_done is high) then
                 goto run_th2
                 else
                 set next_thread to run_th2
                 goto location contained in pc of ft1
                 end if
run_ft2
.
run_ftn
```

Figure 7.9: FSM scheduler

The order in which threads are executed can have a large impact on the system performance. A fixed execution order in every global tick will not be ideal for many applications since data dependencies among threads may not be static. On the other hand, finding an optimal schedule could be greatly complicated by a distribution of threads on multiple processors. This could be a topic for further exploration.
7.2.5 Master tick handler

The tick handler is different for master and slave processors. We first consider the tick handler for the master processor. The template describing its function is shown in Figure 7.10. The point in the code marked by tick handler is arrived at when an FT finishes its local tick. thread_done variable indicates which threads have finished their local ticks. If all bits in thread_done are not high the processor tick end has not yet been reached. The control moves to the scheduler which runs the next thread. Otherwise, the next task is to perform actions at the tick end.

The master processor first has to wait for all slave processors to complete their ticks. When the Tick_finished signal has been set high by every slave processor the global tick is completed. At this point the master tick handler can deal with signals related to rendezvous channels and STs. However, it may have to wait first for one or more STs to complete their iterations. This is discussed later in this section. All Rendezvous_done signals from the previous tick are cleared. Then, the Rendezvous_ready signals from both sides (FT and ST) are checked for each rendezvous channel. If both sides are ready data is transferred between the shared memories and Rendezvous_done is set high. ST_stop_request and ST_start_request signals are checked for each ST. If ST_stop_request is high ST_stop signal is set high. ST_start_request causes the same action on ST_start signal. It should be emphasized that when both ST_stop_request and ST_starts_request signals are high, the ST must first be issued a stop command by ST_stop and then a start command by ST_start. The opposite order is never valid.

While the master processor was doing rendezvous and ST related operations in the tick handler section, slave processors were waiting for the Start_tick_end_actions signal. The master processor sends this command once it has finished with rendezvous channels and STs. The command tells each slave processor to update shared variables that are shared by its FTs. When updating is done, a slave processor has to clear all signals that were written in the previous tick. The master processor also has to update shared variables that are shared by FTs executed on different processors. A slave processor acknowledges that it has finished updating shared variables and clearing signals by
tick_handler:  if all FTs finished their ticks (all bits in threads_done high) then
              for i in 1 to _number_of_processors do
                  await Tick_finished(i)
              end for
wait for SDF iterations to make required number of ticks (optional)
for i in 1 to _number_of_rendezvous_channels do
    clear Rendezvous_done(i)
    if FT_rendezvous_ready(i) and ST_rendezvous_ready(i) are both present then
        do data transfer and emit Rendezvous_done(i)
    end if
end for
for i in 1 to _number_of_ST_threads do
    if ST_stop_request(i) is present then
        set high ST_stop(i)
    end if
    if ST_start_request(i) is present then
        set high ST_start(i)
    end if
end for
clear threads_done
set high Start_tick_end_actions for all slave processors
update shared variables
for i in 1 to _number_of_slave_processors do
    await Tick_end_actions_completed(i)
end for
clear all signals except for Rendezvous_done signals
emit Start_tick to all slave processors
goto run th1
else
goto section pointed by next_thread
end if

Figure 7.10: Master tick handler
setting high `Tick_end_actions_completed`. When the master processor receives the acknowledgment from all slave processors, it clears all its signals except for `Rendezvous_done` signals. Then, it starts a new global tick by making a short pulse `Start_tick` signal.

Before dealing with rendezvous channels and other tick-end activities the master tick handler may have to ensure that the current iteration of an SDF graph completes the specified number of ticks. The influence of SDFG speeds on the system behaviour was discussed in section 4.3. Only SDF graphs that communicate with FSMs have to be taken care of.

![Figure 7.11: Control of SDF iteration length](image)

Figure 7.11 illustrates the action of the master tick handler that is needed to control the length of SDF iterations. In this example we assume that it has been specified that each iteration of an SDF graph takes four ticks to complete. At the end of the fourth tick the master tick handler finds that the SDF thread has still not finished the current iteration. Consequently, it has to wait for the end of the iteration before starting tick end actions. Thus, the first iteration effectively stretches the fourth tick. The other form of control is seen in the next four ticks that are longer than the first four. The ST completes its iteration during the seventh tick and becomes blocked waiting for rendezvous. At the end of the seventh tick, the master tick handler ignores the rendezvous ready signal of the ST since the iteration was completed too early with respect to the number of ticks elapsed in the FSM domain. The rendezvous will happen at the end of the eighth tick.
The actions performed by the master tick handler in Figure 7.11 are necessary to guarantee that the system will have exactly one behaviour. When the speed of an SDFG is not controlled, the system behaviour may be sensitive to the relative speeds of the processing elements, as discussed in section 4.3. In that case, it may be necessary to perform HW/SW co-simulation to ensure that the implementation behaves as intended. This is shown in Figure 7.12. At this level of abstraction, verification is much more difficult. For safety critical applications, where the correctness of the final implementation is extremely important, controlling SDFG speeds is preferable. It should be emphasized though that the operations that control SDFG speeds create significant overhead, which results in higher implementation cost. For this reason, the design flow in Figure 7.12 could be more suitable for applications that are highly cost-sensitive and not safety-critical such as various consumer electronics products.

![DFCharts design flow without SDF iteration control](image)

When each SDF iteration takes a certain number of ticks of the FSM clock, the implementation resembles a multirate synchronous system. Distributed implementation
of synchronous specifications created in Esterel [15], Lustre [16] and Signal [17] has been researched extensively. Unlike DFCharts, synchronous languages do not provide an easy trade-off between implementation efficiency and verification effort. In DFCharts design flow, an SDFG may be forced to follow the FSM clock, but this is not compulsory. Communication between SDFGs and FSMs can be asynchronous, which is likely to result in lower implementation cost. On the other hand, synchronous languages do not support asynchronous communication mechanisms. Signal offers some support for asynchrony since it allows unrelated clocks, but creating asynchronous communication mechanisms like rendezvous and FIFO buffer is very difficult in Signal.

7.2.6 Slave tick handler

The slave tick handler is simpler than the master tick handler as can be seen from the template in Figure 7.13.

```
tick_handler:    if all FTs finished their ticks (all bits in threads_done high) then
                    emit Tick_finished
                    await Start_tick_end_actions
                    update shared variables
                    emit Tick_end_actions_completed
                    await Start_tick
                    clear all signals
                    goto run th1
                else
                    goto section pointed by next_thread
                end if
```

Figure 7.13: Slave tick handler

The previous section has showed that the master tick handler has five functions: controlling the length of SDF iterations, moving data across rendezvous channels, starting and stopping STs, updating shared variables and clearing signals. The slave tick handler only has to update local shared variables and clear signals.
7.3 Frequency relay implementation

The specification for the frequency relay consisting of seven FSMs and one SDFG was presented in section 3.2. Each iteration of SDF1 is set to take five FSM ticks since FSM3 takes five ticks on its path that leads back to the initial state S31, where it is ready again to receive data from SDF1.

Among various ways in which FSMs from the frequency relay can be merged, the hierarchical composition of FSM2 (Figure 3.9) and FSM7 (Figure 3.10) is the most useful one. When a single thread is used for implementation of FSM2 and FSM7 instead of two threads, specification-visible signal alld, specification-invisible signals FT_start and FT_stop, and code sections tick exit and tick entry all disappear. On the other hand, there is no significant loss of concurrency. \textit{update()}, which is the longest procedure in FSM2, cannot be invoked while FSM7 is active. The outgoing transitions of S22, which is refined by FSM7, do not call any procedures. Therefore, there would not be any performance gain if FSM2 and FSM7 were implemented with two threads running on different processors. Other FSM compositions are less attractive since they offer both advantages and disadvantages that have to be weighted carefully. For example, if FSM5 and FSM6 (Figure 3.8) are merged, specification-visible signal st disappears but the resulting FSM has eight states (4x2) which is more than the total of six states for FSM5 and FSM6. In the following steps, we will only use the hierarchical composition of FSM2 and FSM7. Thus, there will be six FSMs to partition: FSM1, FSM27, FSM3, FSM4, FSM5 and FSM6.

There are many allocation and partitioning options for the frequency relay. We limit our attention to two so that we can describe them in more detail. SDF1 has to perform intensive calculations involving long loops, which could not be done with a software implementation within the sampling period of the AC signal (125 µsec). Thus, SDF1 is implemented with a functional unit in both options. FSMs are implemented with one ReMIC in the first option, and two ReMICs in the second option. The partitioning for the second option is shown in Figure 7.14. Although the master processor runs a smaller number of FSMs, its execution load is usually not smaller than that of the slave
processor since FSM27 often invokes the lengthy procedure for updating the thresholds. The other reason for this partitioning is that the two groups of FSMs do not communicate by any specification-visible local signals.

Figure 7.14: Partitioning for the second implementation option

Figures 7.15 and 7.16 show the two implementation options after the synthesis step. Note that only the logical flow of data is showed between shared memories. Address and data buses will depend on the implementation technology. The signals labelled n1, n2 and n3 are the switches for controlling the network load. In Figure 7.16 the resolve lines are marked with r as in FT_stop_r. Schedules were made taking into account data dependencies between the FTs. The schedule for the master processor in the first implementation is ft1, ft27, ft3, ft4, ft5, ft6. The schedule for the master processor in the second implementation is ft1, ft27 while the schedule for the slave processor is ft3, ft4, ft5, ft6.

Figure 7.15: Frequency relay implementation with one ReMIC processor
Figure 7.16: Frequency relay implementation with two ReMIC processors

The lengths of the threads for the first implementation in terms of the number of ReMIC instructions are given in Table 7.1. The numbers are identical for the second implementation except for ft1 which has twenty more instructions due to additional synchronization in the system.

<table>
<thead>
<tr>
<th>thread</th>
<th>ft1</th>
<th>ft27</th>
<th>ft3</th>
<th>ft4</th>
<th>ft5</th>
<th>ft6</th>
</tr>
</thead>
<tbody>
<tr>
<td>instructions</td>
<td>73</td>
<td>208</td>
<td>131</td>
<td>119</td>
<td>278</td>
<td>88</td>
</tr>
</tbody>
</table>

Table 7.1: Thread lengths in the first solution

The program and data memory requirements are shown in Table 7.2. For the second solution, the numbers from individual processors (seen in the brackets where the first
number represents the master processor) are added up. The combined data memory is slightly larger for the second solution due to duplication of variables `threads_done` and `next_thread`. The program memory is also larger due to increased code size for tick handlers.

<table>
<thead>
<tr>
<th></th>
<th>program memory (bytes)</th>
<th>data memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>4296</td>
<td>118</td>
</tr>
<tr>
<td>2p</td>
<td>4584 (1776+2808)</td>
<td>122 (42+80)</td>
</tr>
</tbody>
</table>

Table 7.2. Program and data memories

Table 7.3 presents a performance comparison between the two implementation options. Four rows represent four global ticks. FSM transitions are shown by giving source and sink states. The source state is omitted if the FSM has just become active in the current tick. Additional notes are included where applicable. They indicate exactly which transition is taken when there are multiple transitions between two states, or the value of a variable since that can also influence the amount of time consumed. Columns 8 and 9 show the time taken by the first and the second implementation respectively, measured in ReMIC cycles. The last column shows the percentage difference. Clearly, there is a performance/cost trade-off between the two implementations.

<table>
<thead>
<tr>
<th>FSM1</th>
<th>FSM2</th>
<th>FSM3</th>
<th>FSM4</th>
<th>FSM5</th>
<th>FSM6</th>
<th>FSM7</th>
<th>1p</th>
<th>2p</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>S11→S12</td>
<td>→S21</td>
<td>→S31</td>
<td>→S41</td>
<td>→S51</td>
<td>→S61</td>
<td>→S71</td>
<td>933</td>
<td>666</td>
<td>29%</td>
</tr>
<tr>
<td>S12→S12</td>
<td>S23→S21</td>
<td>S33→S34</td>
<td>S41→S42</td>
<td>S51→S51</td>
<td>S61→S61</td>
<td>X</td>
<td>1302</td>
<td>885</td>
<td>32%</td>
</tr>
<tr>
<td>S12→S12</td>
<td>S22→S22</td>
<td>S32→S33</td>
<td>S41→S41</td>
<td>S51→S51</td>
<td>S61→S61</td>
<td>S74→S75</td>
<td>1047</td>
<td>801</td>
<td>23%</td>
</tr>
<tr>
<td>S12→S12</td>
<td>S23→S21</td>
<td>S34→S31</td>
<td>S43→S41</td>
<td>S51→S54</td>
<td>S61→S62</td>
<td>X</td>
<td>1146</td>
<td>798</td>
<td>30%</td>
</tr>
</tbody>
</table>

1. six thresholds updated
2. the value of \( f_3 \) is zero
3. transition 3 taken
4. four thresholds update

Table 7.3: Performance comparisons between two implementations
As mentioned previously FSM3 takes five ticks on its path from S31 to S31. There are four transitions on the path from S31 to S31, but FSM3 has to spend two ticks in S34 as it waits for FSM4 to make the transition from S42 to S43. Calculating exactly the maximum number of ReMIC instruction cycles that these five ticks can take is required to show whether SDF1 will always be ready to receive the input on ch1 in time. However, this task is quite intensive since there are many possible execution paths in the system. The result does not only depend on FSM3 and FSM4 but also on the other FSMs as well, since they all execute in lock-step and hence influence the tick duration. Currently, all FSMs in a DFCharts specification must use the same clock, but this could change in a future development. Apart from the states of the FSMs, values of variables can also have an effect since they can influence computation time inside procedures.

A tool would be needed to provide an accurate, automated analysis. However, a designer can still produce quick, conservative estimates of the initial performance requirements. The longest tick in the system is shown in row 2 of Table 7.3. In that tick, FSM2 updates all six thresholds, FSM3 calculates the frequency status while FSM4 calculates the average rate of change. For the single processor implementation, the tick lasts 1302 instructions. Multiplying this number by five gives 6510. With two processors, this would be reduced to $5 \cdot 885 = 4425$. Finally, the sampling frequency needs to be taken into account. All instructions that comprise the five ticks must be completed within the period of the sampling clock. Otherwise, SDF1 would miss the input on ch1 while waiting for FSM3 to reach S1. For the sampling frequency of 8kHz, the minimum clock frequency of the single ReMIC is $8\text{kHz} \cdot 6510 = 52.08 \text{ MHz}$. For the two processor implementation the minimum clock frequency is $8 \text{kHz} \cdot 4425 = 35.4 \text{ MHz}$.

Of course, these numbers are largely overestimated. The five ticks under consideration could not all be of the maximum duration. However, the results above do provide a useful initial boundary on the processor performance. As for the hardware that implements SDF1, it needs to be able to complete an iteration is slightly less then $1/fs$, where $fs$ is the sampling frequency.
Chapter 8

Conclusion

Design of embedded systems based on formal models of computation has been gaining acceptance as a sound method for dealing with increasing system complexities. While models of computation have been successfully used for control-dominated systems and data-dominated systems, modelling of heterogeneous systems still poses a challenge. In this thesis, we presented DFCharts, a model of computation for heterogeneous embedded systems that consist of control-dominated and data-dominated parts. We demonstrated how DFCharts could be used in embedded systems design by linking it to system level design languages and implementation architecture. Throughout the thesis, we used a realistic heterogeneous embedded system called frequency relay in order to illustrate the concepts.

DFCharts integrates the hierarchical concurrent finite state machines (HCFSM) model with synchronous dataflow (SDF). As in Argos, FSMs are composed using synchronous parallel, hiding, and refinement operators. The fourth operator called asynchronous parallel is used to connect FSMs with SDFGs. The refinement operator also allows the state of an FSM to be refined by an SDFG. The SDFG becomes active when the state is entered, and it gets instantly terminated when the state is left. Due to the application of the asynchronous parallel and refinement operators, behavioural heterogeneity is addressed using both hierarchical and parallel compositions. This contributes to modelling flexibility which is important for capturing the behaviour of a system.
accurately and producing efficient implementations. The asynchronous parallel operator has been designed so that FSM and SDF, two vastly different models, retain their original characteristics as much as possible. Towards this aim, it allows an SDFG to operate at its own speed. An SDFG only has to synchronize with FSMs between two iterations when it is receiving inputs for the next iteration and sending outputs produced during the previous iteration. Besides SDF, any dataflow model with clearly defined iterations and bounded memory can easily be incorporated. Thus, DFCharts is expressive enough to cover a wide range of embedded systems.

The formal semantics of DFCharts was presented in Chapter 4. We described the ordering of events on a single hierarchical level of DFCharts using the tagged signal model (TSM) framework. The TSM semantics of DFCharts closely focuses on data transfer between FSMs and SDGFs. It shows how values in an FSM array variable appear as multiple tokens in an SDFG buffer, when data is transferred across a rendezvous channel. The automata based semantics, which resembles the semantics of Argos, can be used for the global analysis of a DFCharts specification. It expresses the behaviour of a DFCharts specification in terms of a single, flat FSM. This is achieved by representing the operation of each SDFG as an FSM. The abstract ‘SDF FSMs’ can then be merged with the ‘real FSMs’. ‘SDF FSMs’ have large significance in the automata semantics since they allow a single formalism to embrace both FSMs and SDFGs. When a single FSM is obtained, determinism and reactivity can be analysed. Only specifications that satisfy these two properties are correct. In the automata semantics, data transfer across rendezvous channels is not analysed. All that matters is the event that is generated when a rendezvous occurs. Thus, the automata semantics and TSM semantics complement each other – the former looks at the behaviour of a complete system, the latter focuses on communication through rendezvous channels. An important feature of the automata semantics is the use of multicycle FSMs. In a multicycle FSM, transitions are triggered by different clocks. Apart from DFCharts, this concept can be used for describing the semantics of other models where multiple clocks appear.

DFCharts was used in chapter 5 to analyse the ability of SystemC and Esterel to specify heterogeneous embedded systems. We examined the level of support that the two
languages provide for features that are necessary to capture the behaviour of heterogeneous embedded systems and are found in DFCharts such as synchronous events, rendezvous channels, FIFO channels, hierarchy, preemption etc. Some features are completely supported, others are more difficult to describe. Possible modifications were suggested for both languages in their weak areas. For SystemC, it is mainly control-dominated behaviour. For Esterel, the focus is on data-dominated behaviour.

In general, it takes a lot of effort to verify the correctness of a multi-threaded Java design [8] due to deadlocks that are difficult to detect. It may happen that a design performs correctly before it suddenly crashes for no apparent reason. The Java class library described in Chapter 6 provides an opportunity for a more reliable design in Java. It contains classes that enable specifications to be made according to the DFCharts model. Instead of using threads with mutexes, locks and other mechanisms commonly employed in Java that often lead to unpredictable behaviour, the designer specifies FSMs and SDFGs and connects them with synchronous signals and rendezvous channels. The result is a design with clear semantics.

In Chapter 7, we presented a design methodology with a complete design flow from specification to implementation. DFCharts is used for specification while HiDRA, a multiprocessor architecture introduced in Chapter 2, is used for implementation. An important strength of the methodology is that it starts by capturing the behaviour of a system without any reference to implementation. Besides fast verification, this also allows the designer to easily explore various mapping options before the SW/HW synthesis. We laid a foundation for automated synthesis of heterogeneous embedded systems by specifying in detail how DFCharts is executed on HiDRA. Automated synthesis is a key feature that is missing from most system-level design methodologies. It provides a much bigger improvement in design productivity than just raising the initial level of abstraction above the SW/HW boundary and then using manual refinement to obtain a final implementation. A necessary condition for automated synthesis is to use a model of computation with precisely defined semantics. For this reason the design methodology in Chapter 7 is based around DFCharts. The frequency relay case study was used to show the practical application of the methodology. Two
implementation options were presented, which demonstrated the trade-off between performance and cost.

8.1 Future research

Extension towards distributed systems

In its current form DFCharts is hardly capable of handling distributed systems, since it is very difficult to distribute FSMs with synchronous/reactive communication. Even in SoC implementations, problems may be encountered if the control-dominated behaviour is large, consisting of many concurrent FSMs. It may be no longer satisfactory to have only one clock triggering all FSMs. An attractive solution is to create a model that represents a network of DFCharts nodes. Each DFCharts node would have a separate clock for its FSMs. The communication between DFCharts nodes could be performed through rendezvous or FIFO channels. The required changes in the semantics would be straightforward. Only the asynchronous parallel operator would be significantly affected. Besides connecting an FSM and an SDFG, which it currently does, it would also have to be able to connect two FSMs that belong to different DFCharts nodes. Bigger changes would appear in implementation. In the current implementation described in Chapter 7, there is a master processor which coordinates the execution of the whole system. When a DFCharts network is implemented, there will not be a processor that knows the state of every other processor. Local master processors for individual DFCharts nodes could still exist, but it would be inefficient to have a global master processor even if the whole system is implemented on a single chip. If DFCharts nodes communicate by rendezvous, they will be solely responsible for synchronization because of the absence of the global master processor. Furthermore, it is more difficult to implement rendezvous between two FSMs than rendezvous between an FSM and SDFG. An SDFG must complete communication on all rendezvous channels before it can begin the next iteration. When two FSMs are involved, both sides are able to abort communication on the rendezvous channel. In the case of distributed implementation, the rendezvous protocol is additionally complicated by the fact that it
can no longer be assumed that the transmission of data across a channel is instantaneous.

**Strong abort**

As in Argos, only weak abort is available in DFCharts. Strong abort where an FSM is not allowed to produce outputs in the instant of preemption would be a useful addition for modelling control-dominated behaviour. However, it would make handling of rendezvous more difficult. In the current semantics, when two rendezvous states associated with the same channel are reached, the rendezvous has to happen. With strong abort, this assumption would have to be lifted leading to more complicated semantics and implementation.

**Including more expressive dataflow models**

Inputs arrive periodically in most signal process applications. Consequently, data rates are static on all internal and external channels. For such applications, SDF and related static dataflow models can produce very efficient implementations. On the other hand, there are a significant number of applications with variable data rates especially in the multimedia domain. In DFCharts, variability in data rates can be handled to some extent by using several different SDF graphs at run time. However, it would certainly be useful to include a model with dynamic dataflow like Kahn process networks (KPN). Unlike SDF, KPN does not have an iteration. The main issue would be when to exchange data between a KPN and FSMs.

**Program state**

Imperative statements inside a state that can be compiled into an FSM would be a useful addition to the graphical syntax of DFCharts. This could be done in a similar fashion as in Synccharts where Esterel programs can be inserted in states. Instead of Esterel, DFCharts could use SystemJ [63], which is built on top of Java. A DFCharts state could be refined by a SystemJ *reaction*, which is comparable to a module in Esterel. Only reactions that can be compiled into an FSM with datapath should be placed in a
DFCharts specification. Dynamic memory creation should not be allowed. This requirement stems from the semantics of DFCharts. Further research on how to describe SDF and related dataflow models in SystemJ would open up a possibility of having the complete DFCharts model captured in SystemJ.

**Formal verification**

While formal verification for DFCharts has not yet been developed, the formal semantics of DFCharts based on MCFSM represents a large step towards this goal. MCFSM can be used as an efficient input model for formal verification. We demonstrated in chapter 4 that the MCFSM model captures a mixed synchronous/asynchronous system with fewer transitions than the common approach of using a fictitious base clock to read real clocks. However, parallel products in MCFSM can create state explosion as in other models. State explosion may adversely impact the scalability of DFCharts, since it makes verification of complex systems increasingly difficult. Therefore, this issue requires a careful investigation.

**Proof of correctness for DFCharts implementation**

Chapter 7 provides detailed description of how an implementation is produced from a DFCharts specification. The methodology would be further strengthened by providing a proof of equivalence between specification and implementation levels.
References


[18] www.esterel-tecnologies.com


[21] www.mathworks.com


[93] Li Hsien Yoong, Partha Roop and Zoran Salcic, "Compiling Esterel for distributed execution", in Proceedings of Synchronous Languages, Applications and Programming (SLAP’06), March 2006.


