A Clustering-Based MPSoC Design Flow for Data Flow-Oriented Applications

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.

vorgelegt von

Joachim Falk

aus Erlangen
Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg
Tag der mündlichen Prüfung: 21.11.2014

Vorsitzende des Promotionsorgans: Prof. Dr.-Ing. habil. Marion Merklein

Gutachter: Prof. Dr.-Ing. Jürgen Teich
          Prof. Dr.-Ing. habil. Christian Haubelt
          Prof. Dr. Ir. Twan Basten
Acknowledgments

First, I would like to express my sincere gratitude to Prof. Dr.-Ing. Jürgen Teich for his supervision and useful critiques of this research work. Moreover, I want to give my appreciation to Prof. Dr.-Ing. Christian Haubelt and Prof. Dr. Ir. Twan Basten for agreeing to be the co-examiners of this thesis. I also want to thank the other members of the Ph. D. committee, Prof. Dr.-Ing. Michael Glaß and Prof. Dr.-Ing. Sebastian Sattler. I am particularly grateful for the fruitful discussions with my colleagues from the University of Erlangen-Nuremberg and the University of Rostock. In particular, I want to thank Prof. Dr.-Ing. Christian Haubelt, Christian Zebelein, Jens Gladigau, and Martin Streubühr for their assistance in developing solutions to the various challenges encountered during my work on this thesis. My special thanks are extended to Prof. Dr.-Ing. Michael Glaß, Tobias Schwarzer, Stefan Wildermann, and Kerstin Falk for proofreading parts of this thesis. I would also like to recognize Rainer Dorsch for providing the opportunity to validate parts of the presented design flow in an industry-relevant case study.

Last but not least, I wish to acknowledge the invaluable support given by my family while undertaking the presented research work.
**Contents**

1 Introduction 1

2 Data Flow Fundamentals 7
   2.1 Data Flow 7
   2.2 Static Data Flow 9
      2.2.1 Homogeneous Data Flow 10
      2.2.2 Synchronous Data Flow 12
      2.2.3 Cyclo-Static Data Flow 15
   2.3 Dynamic Data Flow 17
      2.3.1 Kahn Process Networks 19
      2.3.2 Boolean Data Flow 21
      2.3.3 Dennis Data Flow 22
      2.3.4 FunState 25
   2.4 Hierarchical Integration of Finite State Machines and Data Flow 27
      2.4.1 Refining Data Flow Actors via Finite State Machines 28
      2.4.2 Refining Finite State Machine States via Data Flow 31

3 SysteMoC 33
   3.1 Electronic System Level Design and SystemC 34
   3.2 Integration of SysteMoC and Design Space Exploration 37
   3.3 The Model of Computation of SysteMoC 44
      3.3.1 SysteMoC Actors 44
      3.3.2 SysteMoC Network Graphs 48
      3.3.3 SysteMoC FIFO Channels 50
   3.4 The SysteMoC Language 50
      3.4.1 Specification of Actors 51
      3.4.2 Specification of the Communication Behavior 52
      3.4.3 Specification of the Network Graph 56
   3.5 Hierarchical SysteMoC Models 57
      3.5.1 Hierarchical Finite State Machines 57
      3.5.2 Hierarchical Network Graphs 61
<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.3</td>
</tr>
<tr>
<td>3.6</td>
</tr>
<tr>
<td>3.7</td>
</tr>
<tr>
<td>3.8</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4.1</td>
</tr>
<tr>
<td>4.2</td>
</tr>
<tr>
<td>4.3</td>
</tr>
<tr>
<td>4.3.1</td>
</tr>
<tr>
<td>4.3.2</td>
</tr>
<tr>
<td>4.4</td>
</tr>
<tr>
<td>4.4.1</td>
</tr>
<tr>
<td>4.4.2</td>
</tr>
<tr>
<td>4.4.3</td>
</tr>
<tr>
<td>4.4.4</td>
</tr>
<tr>
<td>4.4.5</td>
</tr>
<tr>
<td>4.4.6</td>
</tr>
<tr>
<td>4.5</td>
</tr>
<tr>
<td>4.5.1</td>
</tr>
<tr>
<td>4.5.2</td>
</tr>
<tr>
<td>4.6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5.1</td>
</tr>
<tr>
<td>5.2</td>
</tr>
<tr>
<td>5.3</td>
</tr>
<tr>
<td>5.4</td>
</tr>
<tr>
<td>5.4.1</td>
</tr>
<tr>
<td>5.4.2</td>
</tr>
<tr>
<td>5.4.3</td>
</tr>
<tr>
<td>5.5</td>
</tr>
<tr>
<td>5.6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6.1</td>
</tr>
<tr>
<td>6.2</td>
</tr>
</tbody>
</table>

**German Part** | 223

**Bibliography** | 229

**Author’s Own Publications** | 243
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Symbols</td>
<td>247</td>
</tr>
<tr>
<td>Acronyms</td>
<td>253</td>
</tr>
<tr>
<td>Index</td>
<td>255</td>
</tr>
</tbody>
</table>
Due to the rising hardware capabilities of embedded systems, their offered functionality has increased tremendously. However, the comparatively slow improvements in battery technology in the same time frame has caused only a slight relaxation of power and energy constraints for portable devices. Therefore, power-efficient implementations of the requested functionality is strongly desired, leading to heterogeneous implementations using power-efficient dedicated accelerators and multicore processors. Other embedded systems like automotive controller networks are naturally heterogeneous distributed systems. The above mentioned reasons are contributing factors to the situation that, in all likelihood, future embedded systems of medium complexity will no longer consist of a single computation resource, but multiple connected and heterogeneous resources. Hence, languages used for implementing and modeling applications for these future embedded systems need the ability to exploit the parallelism inherent in such an architecture.

“Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.”
— From “The Problem with Threads,” by Edward A. Lee [Lee06]

This fact causes problems for the traditional implementation of embedded systems via sequential programming languages, as these languages handle concurrency by using threads. Data flow, in contrast, is a modeling paradigm well-suited for the modeling of concurrent systems by concurrently executing actors. Thus, data flow is a natural fit for the distributed nature encountered in future embedded systems. This thesis is concerned with providing improvements for design flows targeting such systems by: (1) providing a modeling language with formal underpinnings in data flow modeling, thus, enabling a trade-off between analyzability and expressiveness of the described application; (2) contributing a methodology called clustering that exploits these possible trade-offs to generate efficient schedules for applications containing actors of high analyzability.
and low expressiveness; and (3) enabling the synthesis of the modeled applications via a synthesis back-end that supports multiple targets such as single core software implementations, virtual prototypes of distributed Multi-Processor System-on-Chip (MPSoC) systems, and dedicated hardware implementations. In the following, the key contributions as well as the limitations of the presented work are outlined. Then, an outline of the structure of this work is given.

**Contributions and Limitations**

In this thesis, a design flow for data flow-oriented applications from an executable specification down to a virtual prototype [GHP+09] will be presented. The scope of the work are executable specifications that can be modeled as data flow graphs like brake-by-wire [GTZ12] or torque vectoring [ZGBT13] from the signal processing domain, models from the image processing domain such as, for instance, a Motion-JPEG [KSS+09] decoder or a pedestrian detection system [KFL+13], as well as applications from the packet processing domain like an InfiniBand network controller [FZH+10*, ZFH+10*].

The first key contribution of this work is the SystemC Models of Computation (SysteMoC) modeling language [FHZT13*, FHT06*] for the Electronic System Level (ESL) and its integration into the SystemCoDesigner Design Space Exploration (DSE) design flow [KSS+09*]. This language has strong formal underpinnings in data flow modeling with the distinction that the expressiveness of the data flow model used by an actor is not chosen in advance but determined from the implementation of the actor [FHZT13*, ZFHT08*]. The second key contribution of this work is the clustering methodology [FKH+08*, FZHT13*] that exploits the presence of highly analyzable static actors in the data flow graph even in the presence of dynamic actors. Actors are called static if they conform to the known data flow models Homogeneous (Synchronous) Data Flow (HSDF) [CHEP71], Synchronous Data Flow (SDF) [LM87b] or Cyclo-Static Data Flow (CSDF) [BELP96] as introduced in Chapter 2. The clustering methodology is neither limited to data flow graphs consisting only of static actors, nor does it require a special treatment of these static actors by the developer of the SysteMoC application. Finally, a correctness proof [FZHT13*] of the presented clustering methodology is given. The third key contribution of this work is the synthesis back-end [ZHFT12*, ZFHT12*, ZFS+14*] for the SysteMoC language. The automatic generation of virtual prototypes is required for the evaluation of the clustering methodology by generating implementations and measuring the resulting throughput of the generated implementations.

In order to determine the data flow model used by a SysteMoC actor, the SysteMoC language, in contrast to SystemC, enforces a distinction between communication and computation of an actor. While modeling of the executable specification in SysteMoC is the preferred design entry to specify this data flow
graph, the executable specification can in theory [FZK+11*] also be specified via pure SystemC if certain coding standards are enforced. These coding standards would replace the enforcement by the SysteMoC language of a clear distinction between communication and computation for each actor. Furthermore, the pure SystemC-based design entry must also be modeled as a data flow graph in contrast to the architecture centric models used in industry. These architecture centric models closely correspond to virtual prototypes containing already memories, buses, and other MPSoC architecture components.

This thesis shows how this distinction between communication and computation can be exploited in order to classify a SysteMoC actor into one of the static data flow models (shaded in Figure 1.1) in the hierarchy of data flow model expressiveness. The ability of classification [FHZT13*, ZFHT08*] serves as a distinction between SysteMoC and Ptolemy [EJL+03, HLL+04, Pto14] as well as Ptolemy Classics [BHLM94] where each actor is associated with a director which explicitly gives the Model of Computation (MoC) under which the actor is executed. This is important as the data flow model of SysteMoC is chosen in order to provide great expressiveness. Hence, the analyzability, e.g., with respect to deadlock freeness or the ability to be executed in bounded memory, of a general SysteMoC model is correspondingly low. However, if this expressiveness is not used by a SysteMoC actor, then the classification of an actor might detect this situation and classify the actor into a data flow model of lower expressiveness but higher analyzability. The classification provides only a sufficient criterion if a general SysteMoC actor conforms to one of the static data flow models. This limitation stems from the fact that the problem in general is undecidable.

For modeling at the ESL, the presented work contributes [FHT06*] an efficient representation and simulation of data flow model-based executable specifications. To demonstrate the usage of SysteMoC for real-world applications, the language has also been used [FZH+10*] in an industrial cooperation to model the executable specification of an InfiniBand network controller, which has been used to show the productivity gain that can be achieved by integrating the SysteMoC ESL model into the design flow used by the industry partner. This productivity gain has been achieved by integrating the verification flow for the hardware design of the InfiniBand network controller with the verification flow for the software development of the firmware responsible to control the network controller via usage of the developed SysteMoC ESL model. In order to facilitate modeling of real-world network applications like the InfiniBand network controller, the SysteMoC language must support the NDF model [Kos78].

A key point of the SysteMoC language in contrast to other data flow modeling frameworks like Ptolemy Classics, Ptolemy, and YAPI [dKSvdW+00] is the ability of executable specifications modeled in SysteMoC to act as an input model for DSE [HFK+07*, KSS+09*] as well as to serve as an evaluator [SFH+06*] for
the DSE to evaluate timing and power characteristics of the executable specification for the different design decisions that are to be explored.

The identification of static actors by means of classification enables the second key contribution, the clustering methodology presented in Chapter 4, to refine the data flow graph by replacing islands of static actors by composite actors. Islands of static actors are connected subgraphs of these static actors in the whole data flow graph described by an executable specification modeled in SysteMoC. Later, in the implementation generated by the synthesis back-end, the remaining actors which are not replaced by a composite actor as well as all composite actors are scheduled by a dynamic scheduler at run time. Thus, by combining multiple static actors into one composite actor, the clustering methodology [FKH+08*, FZK+11*, FHZT13*] reduces the number of actors which have to be scheduled by the dynamic scheduler. This reduction is of benefit as it has been shown [BBHL95, BL93, BML97] that compile-time scheduling of static actors [LM87b] produces more efficient implementations in terms of latency and throughput than run-time scheduling of the same actors by a dynamic scheduler.

Furthermore, an improved rule-based representation [FZHT11*, FZHT13*] of the schedule contained in the composite actor is also discussed. The rule-based representation trades latency and throughput in order to achieve a more compact code size of the schedule contained in a composite actor as compared to the automata-based representation used in [FKH+08*, FZK+11*, FHZT13*].
Finally, the synthesis back-end [ZHFT12*, ZFHT12*] generates an implementation by transforming the executable specification modeled in SysteMoC according to a set of design decisions for the implementation. Examples for design decisions are allocation and binding decisions of the DSE tool as well as the schedules generated by the clustering methodology. The allocation and binding decisions define the set of hardware resources which are used in the virtual prototype and the information which SysteMoC actor is implemented on which resource of the virtual prototype. The schedules generated by the clustering methodology are used by the synthesis back-end to reduce the scheduling overhead of the generated software. The presented synthesis back-end contributes to the realization of the promised productivity gain of ESL modeling by providing an automatic way to generate virtual prototypes for SysteMoC applications. Furthermore, the synthesis back-end is used for the realization of synthesis-in-the-loop by the DSE design flow. Synthesis-in-the-loop enables the DSE to evaluate its design decisions based on real implementations.

**Outline**

The structure of this thesis follows the design flow proposed in this work. The fundamentals of data flow modeling are introduced in Chapter 2. This chapter is also used to introduce the basic notation which is used throughout this thesis. Furthermore, the trade-off between the expressiveness and the analyzability of different data flow models is discussed. Based on these discussions, the underlying data flow model of the input language of the proposed design flow is chosen.

The input language—an extension of SystemC—called SysteMoC is introduced in Chapter 3. The SysteMoC language is used to model the executable specification which is the input for the proposed design flow.

In Chapter 4, a methodology called clustering is introduced. The clustering methodology exploits the derived classification of SysteMoC actors to generate optimized schedules for islands of static actors. The goal of the clustering methodology is the replacement of the static island by a single actor, implementing a schedule for the static actors in the island in such a way that the performance of the whole graph in terms of latency and throughput is optimized.

Subsequently, in Chapter 5, a synthesis back-end will be presented that supports multiple targets such as single core software implementations, virtual prototypes of distributed MPSoC systems, and dedicated hardware implementations.

Finally, a conclusion of the presented work and an outlook on possible future work is given in Chapter 6.
Data flow is a modeling paradigm well-suited for the modeling of concurrent systems by concurrently executing actors. Thus, data flow is a natural fit for the distributed nature encountered in future embedded systems. Furthermore, the design entry for the presented design flow for these future embedded systems is given by an executable specification modeled in the SystemC Models of Computation (SysteMoC) language. This language has strong formal underpinnings in data flow modeling.

In this chapter, a general introduction to data flow is given in Section 2.1. Furthermore, the trade-off between analyzability and expressiveness of these Models of Computation (MoCs) will be discussed. Data flow models can be classified into static data flow models, known from literature for their high analyzability but low expressiveness, and dynamic data flow models, known for their high expressiveness but low analyzability.

First, three important static data flow models Homogeneous (Synchronous) Data Flow (HSDF), Synchronous Data Flow (SDF), and Cyclo-Static Data Flow (CSDF) known in literature will be recapitulated in Section 2.2. Next, in Section 2.3, the dynamic data flow models Kahn Process Network (KPN) as well as its usual realization as Dennis Data Flow (DDF), Boolean Data Flow (BDF), and Functions Driven by State Machines (FunState)—a realization of the Non-Determinate Data Flow (NDF) model—are discussed. Finally, ways to integrate Finite State Machines (FSMs) into data flow models are reviewed in Section 2.4. Note that parts of this chapter are derived from [FHZT13].

2.1 Data Flow

A Data Flow Graph (DFG) [Kah74] is a graph consisting of vertices called actors and directed edges called channels. Whereas the actors are used to model functionality, thus computations to be executed, the channels represent data communication and storage requiring memory to be realized. Channel memory is conceptionally considered to be unbounded, thus representing a possibly infinite amount of storage. Moreover, it is customary [Den74] to separate the
2. Data Flow Fundamentals

A very simple DFG consisting of a channel communicating data produced by actor $a_1$ and consumed by actor $a_2$.

**Figure 2.1**: A very simple DFG consisting of a channel communicating data produced by actor $a_1$ and consumed by actor $a_2$.

computation of an actor into distinct steps. These steps are called actor firings. An actor firing is an atomic computation step that consumes a number of data called tokens from each incoming channel and produces a number of tokens on each outgoing channel. More formally, a DFG is defined as follows:

**Definition 2.1** (Data Flow Graph). A DFG is a directed graph $g = (A, C)$, where the set of vertices $A$ represents the actors and the set of edges $C \subseteq A \times A$ represents the channels. Additionally, a delay function $\text{delay} : C \to V^*$ is given.\(^1\) It assigns to each channel $(a_{\text{src}}, a_{\text{snk}}) = c \in C$ a (possibly empty) sequence of initial tokens.\(^2\) The set $V$ is the set of data values which can be carried by a token.

An example of a very simple DFG according to Definition 2.1 is depicted in Figure 2.1. It consists of two actors $a_1$ and $a_2$ which are connected by a channel $c_1$. A channel has, if not otherwise noted, infinite channel capacity, i.e., it can store an infinite number of tokens that are in transit between the two actors connected by the channel. For notational convenience, two functions are used to refer to the source actor and the sink actor of a channel.

\(^1\)The $^*$-operator is used to denote Kleene closure of a value set. It generates the set of all possible finite sequences of values from a value set, that is $X^* = \cup n \in \mathbb{N}_0 : X^n$. \(\mathbb{N}_0\) denotes the set of non-negative integers, that is \{0, 1, 2, \ldots\}.

\(^2\)In some data flow models that abstract from token values, the delay function may only return a non-negative integer that denotes the number of initial tokens on the channel. In such a context, the number of initial tokens may also be called the delay of a channel.
2.2 Static Data Flow

- The \textbf{src} : \( C \rightarrow A \) function denotes the source actor \( a = \text{src}(c) \) producing tokens onto a channel \( c \), e.g., \( \text{src}(c_1) = a_1 \) for the DFG depicted in Figure 2.1.

- The \textbf{snk} : \( C \rightarrow A \) function denotes the sink actor \( a = \text{snk}(c) \) consuming tokens from a channel \( c \), e.g., \( \text{snk}(c_1) = a_2 \) for the DFG depicted in Figure 2.1.

The channel \( c_1 \) enables a transmission of data values \( \nu_1, \nu_2, \ldots \) from \( a_1 \) to \( a_2 \). Each data value is carried by a \textit{token}. For data flow models, a \textit{token} represents the atomic unit of data production and consumption. Tokens are queued (as exemplified in Figures 2.1b to 2.1c) and de-queued (Figure 2.1d) on a channel in First In First Out (FIFO) order. Between Figure 2.1a and Figure 2.1b a so-called \textit{actor firing} of actor \( a_1 \) has occurred, producing a token with value \( \nu_1 \). Another actor firing, producing an additional token with value \( \nu_2 \), has occurred in Figure 2.1c. While Figures 2.1c and 2.1d depict the state of the DFG after the first actor firing of actor \( a_1 \) has produced the token with value \( \nu_1 \) but before it has been consumed by the first firing of actor \( a_2 \). The state of a DFG is given by the number and values of tokens on each channel as well as, possibly, the internal states of all actors. For analysis purposes of static data flow models, the values of these tokens do not need to be considered. A resulting next state of the DFG after firing of actor \( a_2 \) is not depicted in Figure 2.1, but consists of the DFG where only the token with value \( \nu_2 \) remains on the channel.

An actor is \textit{fireable} (also called \textit{enabled}) if and only if all the tokens the next actor firing would consume are present on the input channels of the actor, e.g., the actor \( a_2 \) is enabled starting from Figure 2.1b as the token with value \( \nu_1 \), which is the only token that will be consumed by firing of \( a_2 \), is present on the channel.

Data flow is a very general MoC. Decades of research \cite{CHEP71, Den74, Kah74, LM87b, Buc93, Par95, BELP96, TSZ99, GLL99, BB01, HB07, PSKB08, GJRB11} into its applications have lead to a multitude of different data flow models. All of them make different trade-offs between their expressiveness and their analyzability, e.g., with respect to \textit{deadlock freeness}, the ability to be executed in \textit{bounded memory}, or the possibility to be \textit{scheduled at compile time}. In the following, the most important data flow MoCs are briefly reviewed, starting with those data flow models with the least expressiveness.

2.2 Static Data Flow

Of primary interest for the expressiveness of a data flow model are so-called \textit{consumption} and \textit{production rates}. The \textit{consumption rate} \( (\text{cons}(c)) \) of an actor firing from a connected channel \( c \) is the number of tokens which are consumed
by that actor from the channel while firing. To exemplify, consider Figure 2.1d where the first firing of actor \(a_2\) consumes one token from channel \(c_1\). Therefore, the consumption rate of the first firing of actor \(a_2\) from channel \(c_1\) is one. The production rate \(\text{prod}(c)\) is defined analogously. That is, the production rate of an actor firing to a connected channel is the number of tokens which are produced by that actor on the channel while firing. To exemplify, consider Figure 2.1b where the first firing of actor \(a_1\) produces one token onto channel \(c_1\). Therefore, the production rate of the first firing of actor \(a_1\) to channel \(c_1\) is one.

An actor is called a static data flow actor if its consumption and production rates neither depend on the values of the tokens which are consumed by the actor nor on the points in time at which tokens arrive on the channels connected to the actor. Therefore, the consumption and production rates depend on some internal state of the actor which is not influenced by the values of the tokens which are consumed by the actor. It must be assured that both the transitions between this internal states as well as the derivation of the consumption and production rates from this internal state are deterministic. Thus, the communication behavior of static actors can be fully predicted at compile time. The consumption and production rates of an actor are even further constrained in well-known static data flow models, which are presented in the following.

### 2.2.1 Homogeneous Data Flow

The simplest data flow model is Homogeneous (Synchronous) Data Flow (HSDF). Data flow graphs corresponding to this data flow model are also known as marked graphs \cite{CHEP71} in literature. The communication behavior of HSDF actors is constrained in such a way that each actor firing must produce and consume exactly one token on, respectively, the outgoing and incoming channels, i.e., \(\forall c \in C : \text{cons}(c) = \text{prod}(c) = 1\).

Due to the low modeling power of the HSDF model, it is also highly analyzable, e.g., with respect to the ability to be executed in bounded memory. To exemplify, the existence of an infinite sequence of actor firings for the actors of an HSDF graph such that the number of queued tokens on any channel of the HSDF graph does not exceed a finite bound is considered. The existence of such an infinite sequence for an HSDF graph guarantees that the graph can be executed in bounded memory. It is proven in \cite{CHEP71} that an HSDF graph has such an infinite sequence of actor firings if and only if each directed cycle of the graph contains at least one initial token. An HSDF graph satisfying this criterion is depicted in Figure 2.2a. The graph contains three directed cycles,

\[\text{In principle, the requirement of independence of the token arrival times can be neglected if sequence determinacy} \ [\text{LSV98}] \ \text{is still assured.}\]
which are depicted in Figure 2.2b - Figure 2.2d, each containing the channel \( c_2 \) carrying the required initial token.

If each directed cycle of the HSDF graph contains at least one initial token, then there exists a finite sequence of actor firings \( a \) that brings the HSDF graph to the same state as it started from, e.g., as depicted in Figure 2.3 for the graph from Figure 2.2a. The sequence of actor firings \( a \) is also called an iteration of the DFG. The length of an iteration is the number of its actor firings, e.g., six actor firings for the iteration \( a = \langle a_1, a_5, a_4, a_6, a_3, a_2 \rangle \).\(^4\) For HSDF, this corresponds exactly to the number of actors in the graph. An infinite repetition of such an iteration, e.g., \( \langle a_1, a_5, a_4, a_6, a_3, a_2 \rangle^* \), is also called a Periodic Static-Order Schedule (PSOS) [DSB\(^+\,13\)] of the graph.

On the other hand, if the initial token is removed from channel \( c_2 \), the actors \( a_2 \) to \( a_6 \) are deadlocked. This is due to the fact that each of the five actors is now part of a so-called delay-less cycle. A cycle is called delay-less if none of its edges are carrying an initial token. Note that the resulting HSDF graph is not

\(^4\)The notation \( \langle x_1, x_2, x_3 \rangle \) is used to denote sequences of values \( x_i \) which can be manipulated via sequence head and tail operations, concatenations, and similar operations.
deadlocked, but an infinite accumulation of tokens on channel $c_1$ is occurring due to the infinite firing of the only fireable actor $a_1$.

### 2.2.2 Synchronous Data Flow

In the Synchronous Data Flow (SDF) [LM87a] MoC the communication behavior of the actors is constrained to have consumption and production rates being constant for all different firings of an actor. Moreover, consumption and production rates for different connected channels are assumed to be arbitrary positive integer constants. Therefore, in SDF, the consumption and production rates can be expressed only by two functions $\text{cons}$ and $\text{prod}$, respectively.

- The $\text{cons} : C \rightarrow \mathbb{N}$ function specifies for each channel $c \in C$ the number of consumed tokens from the channel $c$ by an actor firing of the actor $\text{skn}(c)$.$^5$

$^5$The symbol $\mathbb{N}$ is used to denote the set of natural numbers, that is the set $\{1, 2, 3, \ldots \}$. 
2.2 Static Data Flow

Figure 2.4: Example of a Synchronous Data Flow graph

- The \( \text{prod} : C \rightarrow \mathbb{N} \) function specifies for each channel \( c \in C \) the number of produced tokens onto the channel \( c \) by an actor firing of the actor \( \text{src}(c) \).

It is customary to annotate the consumption and production rates at the beginnings and endings of the channel edges in visual representations of SDF graphs. An example of an SDF graph is depicted in Figure 2.4. Consumption and production rates of one, e.g., \( \text{cons}(c_1) = 1 \), are traditionally not annotated to reduce clutter.

The question if a DFG can be executed in bounded memory is now also considered for SDF graphs. The answer to this question lies in the calculation of an iteration for the SDF graph. It is proven in [LM87b] that such an iteration is unambiguously defined by the consumption and production rates of the actors in the SDF graph.

The iteration is determined by so-called balance equations. Each balance equation corresponds to one channel in the SDF graph. The balance equation for a channel \( c \) is given below:

\[
\eta_{\text{src}(c)} \cdot \text{prod}(c) = \eta_{\text{snk}(c)} \cdot \text{cons}(c)
\]

In the above equation, the variable \( \eta_{\text{src}(c)} \) denotes the number of actor firings of the actor producing tokens onto channel \( c \) while \( \eta_{\text{snk}(c)} \) denotes the number of actor firings of the actor consuming tokens from channel \( c \). With \( \text{prod}(c) \) and \( \text{cons}(c) \) the left and right side of the equation denote the number of tokens that have been produced and consumed by the \( \eta_{\text{src}(c)} \) source actor and \( \eta_{\text{snk}(c)} \) sink actor firings, respectively. For an iteration, both sides of the equation must be equal, otherwise the number of consumed and produced tokens do not balance, thus an iteration would not bring the graph into the same state as it has started from.

To exemplify, the balance equations for the SDF graph depicted in Figure 2.4 are given below:

\[
\eta_1 \cdot 3 = \eta_2 \cdot 1 \quad (2.1)
\]
\[
\eta_2 \cdot 2 = \eta_3 \cdot 3 \quad (2.2)
\]
\[
\eta_3 \cdot 3 = \eta_2 \cdot 2 \quad (2.3)
\]
Each of the three balance equations (cf. Equations (2.1) to (2.3)) correspond to one of the channels \( c_1 \) to \( c_3 \), respectively. The number of firings for actor \( a_1 \) to \( a_3 \) are given by \( \eta_1 \) to \( \eta_3 \). The above balance equations have a solution with \( \eta_1 = 1 \cdot n \), \( \eta_2 = 3 \cdot n \), and \( \eta_3 = 2 \cdot n \), where \( n \) is any positive integer. It is proven in [LM87b] that if there is no solution besides the trivial zero vector, that is \( \eta_1 = \eta_2 = \eta_3 = 0 \), then the SDF graph either deadlocks or cannot be executed in bounded memory. On the other hand, if there exists such a solution, then the SDF graph is called consistent. Then, the SDF graph may still either deadlock or can be executed in bounded memory.

However, firings can only be non-negative integers. Therefore, if there is a solution besides the trivial zero solution, then an \( n \) must be selected that results in non-negative integer values for the variables \( \eta_1 \ldots \eta_3 \). Using the convention established in [BML97], the minimum positive integer solution to the set of balance equations, e.g., here with \( n = 1 \), is called the repetition vector \( \eta_{\text{rep}} \) of an SDF graph.

The SDF graph depicted in Figure 2.4 has a repetition vector of \( \eta_{\text{rep}}^1 = 1, \eta_{\text{rep}}^2 = 3, \eta_{\text{rep}}^3 = 2 \) (or \( \eta_{\text{rep}} = (1, 3, 2) \) for short).

Moreover, the existence of a repetition vector does not guarantee that the SDF graph can be executed iteratively in bounded memory, that is the existence of a repetition vector is a necessary criterion, but not a sufficient one. To exemplify, the initial tokens from the SDF graph depicted in Figure 2.4 are removed. This does not change the calculation or existence of the repetition vector of the SDF graph. However, without any initial tokens, neither actor \( a_2 \) nor actor \( a_3 \) can ever be fired. Therefore, an infinite accumulation of tokens on channel \( c_1 \) may occur assuming an infinite firing of the only fireable actor \( a_1 \).

In contrast, with the initial tokens as depicted in Figure 2.4, an iteration can be realized by the sequence \( \langle a_3, a_1, a_2, a_2, a_3, a_2 \rangle \) of actor firings as shown in Figure 2.5. The length of this iteration is determined by summing all the entries of the repetition vector.\(^7\)

\[
\sum_{n \in \mathcal{I}(\eta_{\text{rep}})} \eta_{\text{rep}}^n
\]

In general, the test for checking whether a computed repetition vector may also execute an iterative deadlock-free schedule is to fire each fireable actor as many times as implied by the repetition vector until the iteration is finished or a deadlock has occurred. Note that in the general case, the length of the iteration may be exponential in the size of the SDF graph. Hence, in contrast to HSDF

\(^6\)To avoid the usage of double subscripts, the technically correct notation \( \eta_{a_n} \) will not be used.

\(^7\)In the following, the function \( \mathcal{I} \) is used to denote the set of indices of a vector, e.g., with \( \eta_{\text{rep}} = (\eta_{\text{rep}}^1, \eta_{\text{rep}}^2, \ldots \eta_{\text{rep}}^m) \) the index set of \( \eta_{\text{rep}} \) is \( \mathcal{I}(\eta_{\text{rep}}) = \{ 1, 2, \ldots m \} \).
models, where the question of an infinite execution sequence in bounded memory can be answered in polynomial time [Gou80], the problem is only solvable in exponential time for SDF graphs [PBL95].

2.2.3 Cyclo-Static Data Flow

An extension of the SDF model is Cyclo-Static Data Flow (CSDF). In the CSDF model, the communication behavior of an actor is constrained to have cyclically varying consumption and production rates between actor firings. The length of this cycle is known as the number $\tau$ of phases of a CSDF actor. An actor firing of a CSDF actor is also known as a CSDF phase. To accommodate the cyclical varying consumption and production rates, the definition of the $\text{cons}$ and $\text{prod}$ function have to be extended to return vectors of consumption and production rates, respectively.

- The $\text{cons} : C \rightarrow \mathbb{N}_0^\tau$ function specifies for each channel $c \in C$ a vector $(n_0, n_1, \ldots, n_{\tau-1})$ of length $\tau$ (the number of phases of the CSDF actor $\text{snk}(c)$). The vector entries correspond to the consumption rates of the actor $\text{snk}(c)$ in its different phases, e.g., in its first phase, this actor consumes $n_0$ tokens from the channel $c$.

- The $\text{prod} : C \rightarrow \mathbb{N}_0^\tau$ function specifies for each channel $c \in C$ a vector $(n_0, n_1, \ldots, n_{\tau-1})$ of length $\tau$ (the number of phases of the CSDF actor $\text{src}(c)$). The vector entries correspond to the production rates of the
actor \( \text{src}(c) \) in its different phases, e.g., in its second phase, this actor produces \( n_1 \) tokens onto the channel \( c \).

An example of a CSDF graph is depicted in Figure 2.6. One can see the annotated vectors for the consumption and production rates. Only actor \( a_1 \) is a degenerated case with only one CSDF phase. This degenerated case is equivalent to an SDF actor with a production of 6 tokens per firing on channel \( c_1 \). The two remaining actors \( a_2 \) and \( a_3 \) are true CSDF actors with 2 and 4 phases, respectively. As can be seen, the vectors of consumption and production rates annotated to an actor have to be of the same length. This length exactly corresponds to the number of CSDF phases \( \tau \) of the actor. Furthermore, in contrast to SDF, it is permissible for some entries of the production or consumption rate vector to be zero. This is used to denote that the actor will not produce or consume any tokens on the channel for phases where the corresponding entry in the vector of production or consumption rates is zero, e.g., the actor \( a_3 \) does not produce any tokens on channel \( c_3 \) in its fourth CSDF phase.

![Figure 2.6: Example of a Cyclo-Static Data Flow graph](image)

The question whether there exists an infinite sequence of actor firings for the actors of a CSDF graph such that the number of queued tokens on any channel of the graph does not exceed a finite bound is now considered. Again, an iteration has to be determined for the CSDF graph first. The number of actor firings in an iteration may again be expressed via the repetition vector. For the purpose of calculating the repetition vector, all CSDF actor can be replaced by SDF actors with consumption and production rates derived by summing all rates in the corresponding vectors of consumption and production rates of the CSDF actors. Therefore, the resulting balance equation for a channel \( c \) is as follows.\(^8\)

\[
\eta_{\text{src}(c)} \cdot \sum_{n \in \mathcal{I}(\text{prod}(c))} \text{prod}(c)(n) = \eta_{\text{snk}(c)} \cdot \sum_{n \in \mathcal{I}(\text{cons}(c))} \text{cons}(c)(n)
\]

\(^8\)In the following, a vector \( x = (x_1, x_2, \ldots x_n) \) is also considered to be a function from its index set \( \mathcal{I}(x) = \{1, 2, \ldots n\} \) to its entries, that is \( x(i) = x_i \).
2.3 Dynamic Data Flow

To exemplify, the obtained balance equations for the CSDF graph depicted in Figure 2.6 are as follows:

\[ \eta_1 \cdot 6 = \eta_2 \cdot 3 \]  
(2.4)

\[ \eta_2 \cdot 5 = \eta_3 \cdot 10 \]  
(2.5)

\[ \eta_3 \cdot 10 = \eta_2 \cdot 5 \]  
(2.6)

The Equation (2.4) represents channel \( c_1 \), Equation (2.5) corresponds to \( c_2 \), and finally Equation (2.6) for \( c_3 \). The variables \( \eta_1, \eta_2, \) and \( \eta_3 \) represent the number repetitions of all phases of a CSDF actor. Therefore, the number of individual firings \( \eta_n^{\text{rep}} \) of a CSDF actor \( a_n \) can be derived by multiplying the number of CSDF phases \( \tau_n \) of the actor \( a_n \) with the number of phase repetitions \( \eta_n \). As is the case for SDF models, the smallest solution of the balance equations uniquely determines the (minimal) repetition vector of the CSDF graph, e.g., for the CSDF graph depicted in Figure 2.6, the repetition vector can be determined as \( \eta_n^{\text{rep}} = (\eta_1^{\text{rep}}, \eta_2^{\text{rep}}, \eta_3^{\text{rep}}) = (\eta_1 \cdot \tau_1, \eta_2 \cdot \tau_2, \eta_3 \cdot \tau_3) = (1 \cdot 1, 2 \cdot 2, 1 \cdot 4) = (1, 4, 4) \). Furthermore, the existence of a valid iteration, e.g., as depicted in Figure 2.7, corresponding to the repetition vector needs to be verified as, like in SDF, the existence of the repetition vector is only a necessary but not sufficient criterion for the existence of the iteration.

2.3 Dynamic Data Flow

In contrast to static data flow models, where the consumption and production rates cannot be influenced by the values of the consumed tokens, dynamic data flow actors can vary their consumption and production rates depending on the history of the consumed tokens and on the token to be consumed. Dynamic data flow is the first introduced data flow model where consumption and production rates depend on the data values being consumed and produced. Hence, some more mathematical notations are required. First, signals are introduced in order to represent the history of tokens being transported over a channel:

Definition 2.2 (Signal). A signal \( s \in S \) is (a possibly infinite) sequence of values carried by the tokens being transported over a channel. The set of all signals \( S \) is the set of all possible finite and infinite sequences \( \mathcal{V}^{**} \) of values from the set of universal values \( \mathcal{V} \).

The number of values in a signal will be given by \#s, e.g., \#\langle 5, 8, 7 \rangle = 3. This number will also be called the length of a signal. Furthermore, to access a

---

9The "**"-operator is used to denote infinite Kleene closure of a value set. It generates the set of all possible finite and infinite sequences of values from the value set, that is \( X^{**} = \cup_{n \in \mathbb{N}_{0,\infty}} : X^n \). Where \( \mathbb{N}_{0,\infty} \) denotes the set of non-negative integers including infinity, that is \( \{0, 1, 2, \ldots \infty\} \).
2. Data Flow Fundamentals

![Graphs](image)

(a) Initial state of the CSDF graph
(b) After firing of actor $a_3$
(c) After firing of actor $a_1$
(d) After firing of actor $a_2$
(e) After firing of actor $a_3$
(f) After firing of actor $a_2$
(g) After firing of actor $a_2$
(h) After firing of actor $a_3$
(i) After firing of actor $a_2$
(j) After firing of actor $a_3$

**Figure 2.7:** Example of an iteration for the CSDF graph from Figure 2.6

certain value for any signal, the signal is considered to be a function $s : \mathcal{I}(s) \rightarrow \mathcal{V}$ from its index set to the set of values. If not further specified, the index set of a signal will always be starting from zero and continue in increments of one up to the index $\#s - 1$ of the last defined value in the signal, e.g., given $s = \langle 5, 8, 7 \rangle$, then $s(0) = 5, s(1) = 8,$ and $s(2) = 7$. Moreover, some notation is required to manipulate signals themselves:

- The $\text{tail}_n : S \rightarrow S$ function discards the first $n$ elements of the signal and returns the tail of the signal, e.g., $\text{tail}_2(\langle 5, 8, 7, \ldots \rangle) = \langle 7, \ldots \rangle$.

- The $\land$-operator is used to concatenate two signals, e.g., $\langle 3, 4 \rangle \land \langle 8, \ldots \rangle = \langle 3, 4, 8, \ldots \rangle$.

For convenience reasons, a set of signals can be bundled into a vector $s$ of signals, e.g., $(s_1, s_2, \ldots s_n) = s \in \mathbf{S}$. Likewise, the $\land$-operator is applied to vectors of signals by pointwise extension, that is $(s_1, s_2, \ldots s_n) \land (s'_1, s'_2, \ldots s'_n) = (s_1 \land s'_1, s_2 \land s'_2, \ldots s_n \land s'_n)$. The set $\mathbf{S}$ itself is used to denote the set of all such vectors of bundled signals.
2.3 Dynamic Data Flow

2.3.1 Kahn Process Networks

Kahn Process Networks (KPNs) are one of the oldest data flow MoCs. The original paper [Kah74] of Kahn used a denotational semantics to describe the behavior of a KPN actor. In this denotational semantics, an actor $a$ is described by a monotonic function $\mathcal{K}_a$.\(^{10}\)

$$\mathcal{K}_a : S^n \rightarrow S^m$$

This function, also called a Kahn function, is mapping (possibly infinite) sequences of tokens $s_i \in S$ on the $n$ actor input ports to (possibly infinite) sequences of tokens $s_o \in S$ on the $m$ actor output ports.

The behavior of a KPN is described by an equation system that represents the connections of these actors to each other. To exemplify, the DFG in Figure 2.8 is considered. The corresponding equation systems is given below:\(^{11}\)

\begin{align*}
(s_1, s_2) &= \mathcal{K}_1(s_1) \\
(s_3) &= \mathcal{K}_2(\langle \nu_{\text{initial}} \rangle \triangledown s_3, s_2)
\end{align*}

The Equation (2.7) models the self-loop on actor $a_1$ while Equation (2.8) models the self-loop on actor $a_2$ including the initial token with value $\nu_{\text{initial}}$. Both equations are coupled via signal $s_2$ that models the channel connecting actor $a_1$ and $a_2$.

Figure 2.8: Example of a Kahn Process Network

\(^{10}\)In fact Kahn functions are required to be Scott-continuous, which is a slightly stronger requirement than pure monotonicity. However, in practice every computer implementable function working on concrete signals is Scott-continuous if it is monotonic. For an in-depth discussion of the subtleties involved, see [LSV98].

\(^{11}\)To avoid the usage of double subscripts, the technically correct notation $\mathcal{K}_{a_n}$ will not be used.
The definition of the Kahn functions \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \) are given next. Here, the notation \( \lambda \) denotes the signal without any values (\( \lambda = \langle \rangle \)), that is \( \#\lambda = 0 \).

\[
\mathcal{K}_1(s) = (s, s)
\]
\[
\mathcal{K}_2(s_a, s_b) = \left\{ \begin{array}{ll}
(s_a(0) + s_b(0)) \wedge \mathcal{K}_2(\text{tail}_1(s_a), \text{tail}_1(s_b)) & \text{if } \#s_a \geq 1 \wedge \#s_b \geq 1 \\
\lambda & \text{otherwise}
\end{array} \right.
\]

(2.9)

(2.10)

As can be seen, the function \( \mathcal{K}_1 \) simply replicates its input signal to both of its outputs. In contrast to this, the function \( \mathcal{K}_2 \) describes a pointwise addition of its input signals \( s_a \) and \( s_b \).

However, if Equations (2.7) to (2.10) are considered without further constraints, then multiple solutions are possible. This is due to the fact that the signal \( s_1 \) is not constraint by Equation (2.7). Hence, the signal \( s_1 \) can take on any value from the set of all signals \( S \). This is an unsatisfactory situation for a denotational description of the KPN from Figure 2.8. From an operational point of view, one would expect the signal \( s_1 \) (corresponding to channel \( c_1 \)) to have a single solution, the empty signal \( \lambda \), that is the signal of length zero. This is due to the deadlock implied by the delay-less self-loop of actor \( a_1 \) via channel \( c_1 \).

It is, however, possible to select from the set of solutions a unique solution that corresponds to the operational point of view. This solution is called the least fixed point of the set of equations. To formally define a least fixed point, a definition of the prefix order \( \sqsubseteq \) for signals is required. The prefix order \( s_1 \sqsubseteq s_2 \) is a partial order that is used to denote that a signal \( s_1 \) is a prefix of another signal \( s_2 \). More formally, \( s_1 \sqsubseteq s_2 \) if and only if \( I(s_1) \subseteq I(s_2) \) and \( \forall i \in I(s_1) : s_1(i) = s_2(i) \).

To exemplify, \( \langle 3, 1 \rangle \sqsubseteq \langle 3, 1, 2 \rangle \sqsubseteq \langle 3, 1, 2, \ldots \rangle \), but \( \langle 3, 1 \rangle \nsubseteq \langle 4, 1, 2 \rangle \) as well as \( \langle 3, 1 \rangle \nsubseteq \langle 3 \rangle \).

For convenience reasons, the prefix order is extended to vectors \( s_1, s_2 \in S \) of signals. A vector \( s_1 \) is a prefix of another vector \( s_2 \) if all corresponding vector elements are prefixes of each other, that is \( s_1 \sqsubseteq s_2 \) if and only if \( I(s_1) = I(s_2) \) and \( \forall i \in I(s_1) : s_1(i) \sqsubseteq s_2(i) \).

Assuming that \( S_{\text{solution}} \) is the set of all solutions for the Equations (2.7) to (2.10), then the least fixed point \( S_{\text{lfp}} \) is the least element in the set of solutions, that is \( S_{\text{lfp}} \in \{ s \in S_{\text{solution}} \mid \forall s' \in S_{\text{solution}} \setminus \{s\} : s' \nsubseteq s \} \). However, the question remains if the least fixed point for such a set of equations is unique. Fortunately, it has been proven that, assuming all functions in the set of equations are monotonic, then the least fixed point of such a set of equations either exists and is unique or does not exist at all [DP90]. The monotonicity requirement for the functions corresponds to the customary definition of monotonicity with the prefix order \( \sqsubseteq \) as its ordering relation, that is a function \( \mathcal{K}_\alpha : S^n \rightarrow S^m \) is monotonous if and only if \( \forall s_1, s_2 \in S^n : (s_1 \sqsubseteq s_2) \implies \mathcal{K}_\alpha(s_1) \sqsubseteq \mathcal{K}_\alpha(s_2) \).
A constructive way to find the least fixed point starts with the empty signal \( \lambda \) for all signals. To exemplify, the KPN depicted in Figure 2.8 and its corresponding set of Equations (2.7) to (2.10) are considered. The three signals \( s_1, s_2, \) and \( s_3 \) that correspond to the three channels depicted in Figure 2.8 are initialized to the empty signal \( \lambda \).

\[
\mathbf{s}^0 = (s_1^0, s_2^0, s_3^0) = (\lambda, \lambda, \lambda) \tag{2.11}
\]

The empty signal denotes that at the beginning of the fixed-point computation no token values have been produced onto a channel. Note that the initial token \( \nu_{\text{initial}} \) depicted on channel \( c_3 \) is already considered by Equation (2.8).

The set of equations describing a KPN is used to calculate an updated vector of signals \( \mathbf{s}^n = (s_0^n, s_1^n, \ldots s_m^n) \) from the signals \( \mathbf{s}^{n-1} \) in the previous step. For the KPN depicted in Figure 2.8, the vector of signals \( \mathbf{s}^n = (s_1^n, s_2^n, s_3^n) \) is computed by recursively applying Equations (2.7) to (2.8) to \( \mathbf{s}^{n-1} \) starting from the initial vector of signals given in Equation (2.11).

\[
(s_1^n, s_2^n) = \mathcal{K}_1(s_1^{n-1}) \quad (s_3^n) = \mathcal{K}_2((\nu_{\text{initial}}) \triangleleft s_3^{n-1}, s_2^{n-1})
\]

Due to the monotonicity of the functions in the set of equations, the signals in the previous step must always be a prefix of the updated set of signals, that is \( \mathbf{s}^{n-1} \subseteq \mathbf{s}^n \). This fact prevents the existence of a cycle in the sequence \( \mathbf{s}^0, \mathbf{s}^1, \ldots \mathbf{s}^n \) of updated signal vectors, thus guaranteeing the existence of a unique least fixed point if no contradictions are encountered in the set of equations. This step is repeated until a fixed point has been reached. As can be seen, for the example from Figure 2.8 the initial state with all signals empty is already the least fixed point of the Equations (2.7) to (2.8).

\[
s_1^0 = s_1^1 = \lambda \quad s_2^0 = s_2^1 = \lambda \quad s_2^0 = s_2^1 = \lambda
\]

The question whether there exists an infinite sequence of actor firings for the actors of a KPN such that the number of queued tokens on any channel of the graph does not exceed a finite bound is now considered. Unsurprisingly, this problem is undecidable. This is due to the fact that Kahn functions themselves are general recursive functions, and therefore are Turing equivalent [Ros65].

### 2.3.2 Boolean Data Flow

More surprisingly, a KPN model remains Turing equivalent [Buc93], hence the problem of execution in bounded memory undecidable, if the Kahn functions are restrictions to perform simple arithmetic operations as well as the two primitive switch \( (K_{\text{swt}}(s_1, s_2) = (s_{\text{out,1}}, s_{\text{out,2}})) \) and select \( (K_{\text{sel}}(s_1, s_2, s_3) = s_{\text{out}}) \) functions. A KPN model with these restrictions is also known as a Boolean Data...
2. **Data Flow Fundamentals**

Flow (BDF) graph. The BDF MoC realizes the *switch* and *select* functions via dedicated actors. The simple arithmetic operations are supported by BDF via its ability to model static actors, e.g., like an SDF actor implementing an addition. The *switch* actor, depending on the truth value of a control token from its input signal $s_1$, forwards a token from its input signal $s_2$ to its first or second output. The *select* actor acts in reverse. Depending on the truth value of a control token from signal $s_1$, it either forwards a token from input signal $s_2$ or $s_3$ to its output.

The usage of these two functions together with the arithmetic primitives enable the construction of arbitrary control flow logic. The channels (of infinite capacity) can be used to represent an infinite memory. Together with the control flow logic, a Turing machine can be implemented by the BDF model [Buc93].

### 2.3.3 Dennis Data Flow

While the denotational description of an actor is clear from its Kahn function $\mathcal{K}_a$, an implementation of this functionality is usually not based on a function mapping infinite sequences to infinite sequences, but uses the operation semantics of Dennis Data Flow (DDF) [Den74]. To exemplify, the operational implementation (cf. Example 2.1) of the Kahn actor $a_2$ is considered. As the actor is considered in isolation, the notion of channels is no longer appropriate to identify the source and destination of tokens. Hence, the notion of *ports* is used to identify the source and destination of consumed and produced tokens. A depiction of the actor with its input and output ports has already been given in Figure 2.8, see also Figure 2.9.

In the following, the notion of ports will be used interchangeably with the channels/signals connected to these ports. Hence, expressions like $\text{cons}(i)$, $\text{prod}(o)$, and $\#i$ are used and refer to the consumption rate, production rate, and number of tokens on the channel connected to the respective input or output port.

![Image of actor $a_2$ with input and output ports]

**Figure 2.9:** The KPN actor $a_2$ from Figure 2.8 considered in isolation

---

12In fact, this implementation preference is so strong, that KPN actors are usually understood to be constrained to the subset that can be implemented by the operational semantics of DDF.
2.3 Dynamic Data Flow

For the KPN in Figure 2.8, the input port $i_2$ simply corresponds to the signal $s_2$. In the operational view, this is represented by the channel $c_2$. This channel is queuing the tokens produced by actor $a_1$ in FIFO order. The self-loop from output port $o_1$ to input port $i_1$ is more complex. In the operational view, it is represented by channel $c_3$. The channel is a FIFO containing the initial token $\nu_{initial}$. This is represented in the denotational view (cf. Equation (2.8)) by the temporary signal $s'_3 = \langle \nu_{initial} \rangle \triangle s_3$.

The Kahn function $K_2$ from Equation (2.10) is implemented by a DDF [Den74] actor as depicted in Example 2.1. The DDF model uses (blocking) read (?) and write (!) primitives for its implementation. As is the case for $K_2$, the operational implementation needs one token from input port $i_1$ and one token from input port $i_2$ to produce a token on the output port $o_1$. As the read primitive is blocking, the read operation on port $i_2$ will only be executed after a token from port $i_1$ was available. In the following, $t$ and $f$ are used to denote Boolean truth values.

**Example 2.1** Operational implementation of the Kahn function $K_2$

1: procedure $K_2(i_1, i_2, o_1)$
2:   while $t$ do ∴ Infinite iteration
3:     $i_1$ ? $\nu_1$ ∴ Read a value from the channel attached to port $i_1$
4:     $i_2$ ? $\nu_2$ ∴ Read a value from the channel attached to port $i_2$
5:     $o_1$ ! $\nu_1 + \nu_2$ ∴ Write the sum to the channel attached to port $o_1$

However, the blocking semantics of the DDF operational paradigm can only implement sequential functions [KP78]. While any sequential function is a Kahn function [Lee97], the reverse is not true [Win93]. One could argue that in the real world, the restriction of KPN to the operational semantics of DDF is permissible. However, this raises the problem of *compositionality*. A data flow model is *compositional* if it is possible to take an arbitrary connected subgraph and represent it as an actor in the data flow model. If a data flow model is compositional, then the highly desirable operation of abstraction becomes seamlessly possible. An abstraction operation can be performed by hiding the implementation complexity of an arbitrary connected subgraph of the model behind the interface of an actor. If the data flow model is compositional, then this *composite actor*, which represents the subgraph, can be handled like any other actor in the system. Otherwise, the composite actor is always an exception and needs special treatment.

To exemplify, consider the data flow subgraph depicted in Figure 2.10. In this subgraph, the actors $a_3$ and $a_4$ represent HSDF actors. All HSDF actors can also be represented by DDF actors. The corresponding Kahn function $K_3 \equiv K_4 \equiv K_{3,4}$ is given in Equation (2.12). The topology of the subgraph can be represented by the Equations (2.13) and (2.14).
2. Data Flow Fundamentals

Figure 2.10: A data flow subgraph consisting of two HSDF actors

\[ \mathcal{K}_{3,4}(s_a, s_b) = \begin{cases} \langle s_a(0) \rangle, \langle s_b(0) \rangle & \text{if } \#s_a \geq 1 \land \#s_b \geq 1 \\ \langle \lambda, \langle \lambda \rangle \rangle & \text{otherwise} \end{cases} \quad (2.12) \]

\[(o_1, s_1) = \mathcal{K}_3(i_1, \langle \nu_{i,2} \rangle) \quad (2.13)\]

\[(o_2, s_2) = \mathcal{K}_4(i_2, \langle \nu_{i,1} \rangle) \quad (2.14)\]

It turns out that such a system has a least fixed point for any input signal \(i_1\) and \(i_2\) \cite{LSV98}, and this fixed point again represents a Kahn function. In that sense, the KPN model is compositional. Another important characteristic emerges from this fact. The behavior of a KPN model, be it a complete graph or a subgraph, is independent from the scheduling of actors, which is given by the operational implementation. Such a data flow model is called sequence determinate \cite{LSV98}. However, the resulting Kahn function is not generally representable via the operational semantics of DDF. Therefore, DDF is non-compositional. To exemplify, the Kahn function \(\mathcal{K}_y\) for the least fixed point of the equation system Equation (2.12) to Equation (2.14) is given below.

\[ \mathcal{K}_y(i_1, i_2) = \begin{cases} \langle i_1(0), \lambda \rangle \mathcal{K}_y(\text{tail}_i(i_1), i_2) & \text{if } \#i_1 \geq 1 \\ \langle \lambda, \langle i_2(0) \rangle \rangle \mathcal{K}_y(i_1, \text{tail}_i(i_2)) & \text{if } \#i_2 \geq 1 \\ \langle \lambda, \langle \lambda \rangle \rangle & \text{otherwise} \end{cases} \quad (2.15) \]

where \(\mathcal{K}_y(i_1, i_2) = \begin{cases} \langle i_1(0), \lambda \rangle \mathcal{K}_y(\text{tail}_i(i_1), i_2) & \text{if } \#i_1 \geq 1 \\ \langle \lambda, \langle i_2(0) \rangle \rangle \mathcal{K}_y(i_1, \text{tail}_i(i_2)) & \text{if } \#i_2 \geq 1 \\ \langle \lambda, \langle \lambda \rangle \rangle & \text{otherwise} \end{cases} \quad (2.16)\]

where \(\mathcal{K}_y(i_1, i_2) = \begin{cases} \langle i_1(0), \lambda \rangle \mathcal{K}_y(\text{tail}_i(i_1), i_2) & \text{if } \#i_1 \geq 1 \\ \langle \lambda, \langle i_2(0) \rangle \rangle \mathcal{K}_y(i_1, \text{tail}_i(i_2)) & \text{if } \#i_2 \geq 1 \\ \langle \lambda, \langle \lambda \rangle \rangle & \text{otherwise} \end{cases} \quad (2.17)\]

The problem is caused by the requirement to either serve input \(i_1\) or input \(i_2\) (cf. Equation (2.15)). Obviously, another operational semantics for KPN models is required. In the following, the FunState model is introduced, which, amongst others, can implement \(\mathcal{K}_y\).
2.3 Dynamic Data Flow

2.3.4 FunState

In contrast to the DFG defined in Definition 2.1, Functions Driven by State Machines (FunState)\textsuperscript{13} [TSZ+99] are bipartite graphs of functions and channels. This enables the FunState model to have potentially multiple readers and writers for a channel. The activation of functions is controlled by an FSM. Functions can be annotated with execution times. Note that the FunState FSM only triggers an execution of a function, but does not wait for its completion. Hence, input tokens are consumed by the function when it is triggered by the FSM but output tokens are not guaranteed to have been produced after the FSM transition has been taken. More formally, the definition of a FunState network is given below:

**Definition 2.3** (FunState Network [TTN+98]). A network \( g = (\mathcal{F}, C, E) \) (of a basic FunState model) contains a set of storage units \( C \), a set of functions \( \mathcal{F} \), and a set of directed edges \( E \subseteq (C \times \mathcal{F}) \cup (\mathcal{F} \times C) \). A FunState network \( g \) is a directed bipartite graph. Data flow is represented by moving tokens. The marking \( m \) is the distribution of tokens to storage units. The marking \( m_0 \) is the initial marking.

The FunState model does not only support FIFO channels for its storage units, but also registers. The problem of an operational implementation of \( \mathcal{K} \) by the DDF model is bypassed by the ability of the FunState FSM to wait for conditions on multiple input channels. However, due to this relaxation compared to the DDF model, FunState models in general are no longer determinate [LSV98], that is the output of a FunState model may depend on the scheduling strategy employed to execute it. Hence, FunState is a representative of the Non-Determinate Data Flow (NDF) MoC [Kos78]. More formally, the definition of a FunState FSM is given below:

**Definition 2.4** (FunState FSM [TSZ+99]). A FunState FSM \( \mathcal{R} = (Q, T, q_0) \) (of a basic FunState model) consists of a set of named states \( Q \) and a set of transitions \( T = Q \times Q \times \mathcal{F}_{\text{condition}} \times \mathcal{F} \) and a unique initial state \( q_0 \in Q \). \( \mathcal{F}_{\text{condition}} \) is the set of possible state transition conditions. In the basic model, a condition \( k \in \mathcal{F}_{\text{condition}} \) is a logical propositions over (in)equalities of the form: \\
\text{<query of storage unit>} <relational operator> <constant>
For example, \#c \geq 1 evaluates to true if the channel \( c \) contains at least one token.

However, the FunState model, as introduced above, is not compositional, as it only possesses a central FSM. To obtain compositionality, a hierarchical

\textsuperscript{13}The model was initially published [TTN+98] as State Machine Controlled Flow Diagrams (SCF), but in later literature it is referred to as FunState, which is a short for Functions Driven by State Machines.
FunState model is used. In this model, an FSM can be associated with a subgraph and the subgraph is represented by a vertex in its containing graph. To allow the subgraphs to be self-contained, a notion of ports is again used. These ports are used interchangeably with the channels to which they are connected. To exemplify, the FunState network from Figure 2.11 is considered. This network corresponds to the KPN subgraph from Figure 2.10. Both actors \( a_3 \) and \( a_4 \) from Figure 2.10 are represented by subgraphs in Figure 2.11, but these subgraphs themselves act as functions in the containing graph \( a_\gamma \). The FSM of actor \( a_3 \) activates the function \( f_1 \) if the channels connected to the ports \( i_1 \) and \( i_2 \) of actor \( a_3 \) have at least one token available. The activation of the FSM of actor \( a_3 \) and \( a_4 \) is itself controlled by the FSM of the composite actor \( a_\gamma \). By disregarding the FunState network inside of actor \( a_\gamma \) and only considering the FSM of \( a_\gamma \), the following three operations are possible:

- If the FSM of \( a_\gamma \) is an equivalent to the behavior of the contained FunState network of \( a_\gamma \), then a composition operation has been performed. The implementation complexity of the contained subgraph of \( a_\gamma \) is hidden behind the actor interface of \( a_\gamma \).

- If the FSM of \( a_\gamma \) is simplified in such a way that it implements a superset of the behavior of the contained FunState network of \( a_\gamma \), then an abstraction
2.4 Hierarchical Integration of Finite State Machines and Data Flow

The behavior has to be a superset, as later analysis steps will enforce a property for all possible behaviors of a DFG. This analysis will use the FSM of $a_\gamma$ as the description for the contained subgraph. Therefore, enforcement of a property for all possible behaviors, even artificial behaviors not occurring in the real implementation, is a safe operation as it implies enforcement of the property for the behaviors occurring in real implementation of the system.

- If the FSM of $a_\gamma$ is a subset to the behavior of the contained FunState network of $a_\gamma$, then a refinement step has been performed. Refinements steps have to be reflected in the real implementation of the systems, otherwise analysis steps on the refined FSM will not consider all possible behaviors of the system.

For a formal definition of the compositional FunState model, see [TSZ+99, TTN+98].

2.4 Hierarchical Integration of Finite State Machines and Data Flow

One of the first modeling approaches integrating Finite State Machines (FSMs) with data flow models is *charts (pronounced "star charts") [GLL99, Pto14]. The concurrency model in *charts is not restricted to be data flow: Other choices are discrete event models or synchronous/reactive models. However, in the scope of this section, the discussion has been limited to the FSM/data flow integration. First, a formal definition of a deterministic FSM will be given:

**Definition 2.5 (FSM).** A deterministic FSM is a five tuple $(Q, \Sigma, \Delta, \sigma, q_0)$, where $Q$ is a finite set of states, $\Sigma$ is a set of symbols denoting the possible inputs, $\Delta$ is a set of symbols denoting possible outputs, $\sigma : Q \times \Sigma \to Q \times \Delta$ is the transition mapping function, and $q_0 \in Q$ is the initial state.

In one reaction, an FSM maps its current state $q_{src} \in Q$ and an input symbol $a \in \Sigma$ to a next state $q_{dst} \in Q$ and an output symbol $b \in \Delta$, where $(q_{dst}, b) = \sigma(q_{src}, a)$. Given a sequence of input symbols, an FSM performs a sequence of reactions starting in its initial state $q_0$. Thereby, a sequence of output symbols in $\Delta$ is produced.

FSMs are often represented by state transition diagrams. In state transition diagrams, vertices correspond to states and edges model state transitions, see Figure 2.12. Edges are labeled by guard/action pairs, where guard $\in \Sigma$ specifies the input symbol triggering the corresponding state transition and action $\in \Delta$.
specifies the output symbol to be produced by the FSM reaction. The edge without source state points to the initial state.

One reaction of the FSM is given by triggering a single enabled state transition. An enabled state transition is an outgoing transition from the current state where the guard matches the current input symbol. Thereby, the FSM changes to the destination state of the transition and produces the output symbol specified by the action. To simplify the notation, each state is assumed to have implicit self transitions for input symbols not used as guards for any explicit transition. The action in such cases is empty or the output symbol $\epsilon \in \Delta$ indicating the absence of output.

FSMs as presented in Definition 2.5 can be easily extended to multiple input signals and multiple output signals. In such a scenario, input and output symbols are defined by subdomains, i.e., $\Sigma = \Sigma_1 \times \cdots \times \Sigma_n$ and $\Delta = \Delta_1 \times \cdots \times \Delta_m$.

The *charts approach allows for nesting of DFGs and FSMs by allowing the refinement of a data flow actor via FSMs and allowing an FSM state to be refined via a DFG. First, the refinement of a data flow actor by an FSM will be discussed.

### 2.4.1 Refining Data Flow Actors via Finite State Machines

To use an FSM as a refinement of a data flow actor, the consumption and production rates have to be determined from the FSM for the refined actor. However, the derivation of the consumption and production rates depends on the MoC of the DFG containing the actor. The tokens on the channels connected to the refined data flow actor are mapped to the input alphabet $\Sigma$ of the refining FSM. In the guard of the FSM, the notation $i[n]$ refers to the $n$th token on the channel connected to the input port $i$. In the action of the FSM, the notation $o[n]$ refers to the $n$th token to produce on the channel connected to the output port $o$. Each unique $i[n]$ has a symbol subdomain $\Sigma_m$ associated with it and the cross product of these subdomains forms the input alphabet to the FSM. The same is true for $o[n]$ and the output alphabet $\Delta$, thus perfectly matching the FSM model (see Definition 2.5). In each reaction of the FSM, the corresponding action emits a symbol from the output alphabet of the FSM, which is used to derive a token value to be produced on each outgoing edge of the actor. However, the semantics of the FSM changes depending on the MoC of the DFG the FSM is embedded into.
2.4 Hierarchical Integration of Finite State Machines and Data Flow

- Finite State Machine in Kahn Process Network:
  If an FSM refines a DDF actor, then each state of the FSM is associated with a number of tokens that will be consumed by the next firing of the actor. The semantics is as follows: For a given state \( q \) of the FSM, the number of consumed tokens is determined by examining each outgoing transition of the state. For each input port \( i \) of the actor, the maximum index \( n \) is determined with which the input port \( i \) occurs in any of the transitions leaving the state \( q \). If the input port \( i \) does not occur in any transition, an index of \(-1\) is assumed. Then, for the given state \( q \), the consumption rate for each input port \( i \) is the maximum index \( n \) incremented by one. Note that this derivation of the consumption rates implicitly produces actors which are continuous in Kahn’s sense [Kah74] and can be implemented by a DDF actor.

To illustrate the above concept, consider state \( q_0 \) in the FSM depicted in Figure 2.13 refining the actor \( a_2 \). In this case, regardless if the transition to \( q_1 \) or \( q_2 \) is taken, the actor will consume exactly two tokens from \( i_1 \) and one token from \( i_2 \).

![Figure 2.13: Refinement of DDF actor \( a_2 \) by an FSM in *charts](image)

- Finite State Machine in Synchronous Data Flow:
  When an FSM refines an SDF actor, it must externally obey SDF semantics, i.e., it must consume and produce a fixed number of tokens in each firing. Consider Figure 2.14 taken from [GLL99] as an example. Let us assume that the depicted data flow graph contains only HSDF actors, then the minimal repetition vector is \( \eta_1^{\text{rep}} = \eta_2^{\text{rep}} = 1 \). Since the channel between the actors \( a_1 \) and \( a_2 \) contains no initial tokens, actor \( a_1 \) fires before actor \( a_2 \) in each iteration. In the first iteration, actor \( a_1 \) fires if at least one token is available on each input edge. Afterwards, HSDF actor \( a_2 \) is enabled in the first iteration but also HSDF actor \( a_1 \) might be enabled in the second iteration assuming sufficient input data is available. Hence, both FSMs can execute concurrently.
In the following, the refinement of SDF actors by FSMs will be considered. The refined SDF actor must externally obey SDF semantics. Hence, the consumption and production rates for each port of the refined SDF actor must be determined from the maximum of the consumption and production rates that is exhibited in all transitions of the FSM refining the SDF actor. In the simplest case, it holds that for every port the consumption or production rate of that port is identical in all transitions of the FSM. An example of such an FSM can be seen for the refinement of the SDF actor \( a_2 \) in Figure 2.14. The FSM neatly corresponds to the SDF model consuming one token and producing one token in each reaction.

However, the FSM for actor \( a_1 \) from Figure 2.14 does not correspond to the simple case. In this case, the consumption or production rate of a port of the refined SDF actor is determined from the maximum of the consumption or production rate of this port for every transition of the FSM. Consumed tokens which are not used by the FSM are simply discarded. If the values for the tokens that must be produced according to the production rates of the SDF actor are not provided by the FSM transition, then the so-called default value \( \epsilon \) will be used for the produced token. This solution is a little unsatisfactory and indeed there is the Heterochronous Data Flow (HDF) approach [GLL99] that trades compositionality for the ability to have state-dependent consumption and production rates.

• Finite State Machine in Heterochronous Data Flow: The idea of HDF is similar to parameterized data flow modeling [BB01] and scenario-aware data flow modeling [TGB+06] where changes of consumption and production rates cannot be performed at arbitrary points in time, but only after an iteration of the DFG has finished. Here, an iteration represents a scenario, and after a scenario has been finished, the DFG can switch to a new scenario with possibly changed consumption and production rates. Parameterized data flow models and HDF models differ in the ability of parameterized data flows to express an infinite number of scenarios while HDF models can only express a finite number. For the finite case, a compile-time analysis, e.g., with respect to throughput [TGB+06, SGTB11][GFH+14*], required buffer capacities, deadlock freeness, etc., can be performed.
In HDF \cite{GLL99}, all (actor) FSMs in the HDF graph are forced to only change state once the HDF graph has executed a full iteration. After the iteration is finished, the HDF actors are free to update their state leading to new consumption and production rates for the HDF actors in the system. With these new consumption and production rates, a new balance equation may be solved and a new repetition vector calculated which is executed in the next iteration. For the duration of this next iteration, all HDF actors have to keep their consumption and production rates unmodified. Thus, a product FSM can be computed for a HDF graph where each state represent a scenario and the transitions correspond to the possible switches from one scenario to another one. This product FSM corresponds to the FSM of the FSM scenario-aware data flow model \cite{SGTB11} that is used to represent scenarios (states) and scenario switches (transitions).

Let us consider again Figure 2.14. But now instead of an SDF domain, an HDF domain is assumed. In the case that actor $a_1$ is in state $q_0$, to execute a full iteration of the HDF graph, the actors $a_1$ and $a_2$ are executed exactly once. Note that while actor $a_1$ is executed, it remains in state $q_0$ regardless of the value $i_1[0]$ of the first token. After the full iteration of the HDF graph ($\eta_1^{\text{rep}} = \eta_2^{\text{rep}} = 1$) has finished, the FSM of actor $a_1$ may change its state to $q_1$ if the value $i_1[0]$ of the first token on input port $i_1$ evaluates to true. In the case that actor $a_1$ is now in state $q_1$, a full iteration of the HDF graph corresponds to the sole execution of the actor $a_1$. After the full iteration of the HDF graph has finished, the FSM of actor $a_1$ may change its state to $q_0$ depending on the value $i_2[0]$ of the first token on input port $i_2$.

### 2.4.2 Refining Finite State Machine States via Data Flow

Previously, it has been seen how an FSM can be used to refine a data flow actor. On the opposite side, an FSM can be used to coordinate between multiple DFGs. This coordination is achieved by refining FSM states by DFGs. The DFG is composed into a single actor which is executed if the FSM is in the refined state. To refine a state by a DFG, a notion of *iteration* is necessary as the execution of one reaction of the FSM has to terminate. An *iteration* has been chosen as a natural boundary to stop the execution of the embedded DFG. However, the existence of a finite iteration is undecidable for general DFGs. Hence, the application of refinements of states to DFGs is restricted in *charts to static data flow models, i.e., HSDF, SDF, and CSDF models, which provide such a notion of iteration naturally. Moreover, combining the actors in a subgraph of a DFG into a single composite actor, which will execute an iteration for the subgraph, is not always possible.
As an example, consider Figure 2.15, which is taken from [GLL99]. The top level FSM of the actor $a_1$ consists of two states $q_0$ and $q_1$. If the current state of the FSM is not refined, e.g., $q_0$, then the FSM reacts like described in Section 2.4.1. If the current state of the FSM is refined by an SDF graph, e.g., $q_1$, then for a reaction of the FSM exactly as many tokens will be required as for the execution of the iteration of the DFG refining the state of the FSM.

Assuming that the FSM of actor $a_1$ is in state $q_1$, then each activation of $a_1$ will execute the embedded SDF graph for one iteration. If at least one token of the tokens $i_{2[0]}$ and $i_{2[1]}$ evaluates to false, then the embedded SDF graph will execute one iteration, but the FSM will remain in state $q_1$. If the FSM is not embedded in an HDF graph and the tokens $i_{2[0]}$ and $i_{2[1]}$ evaluate both to true, then a state transition to state $q_0$ will be taken after the embedded SDF has finished its iteration.

The exact behavior of the FSM of actor $a_1$ depends on the MoC of the DFG the actor is embedded in. If the FSM is embedded in the DDF or SDF domain, then the FSM executes a transition after the corresponding DFG has finished its iteration. If the FSM is embedded in the HDF domain, then this transition is delayed until all parent graphs have finished their iteration.

In the next chapter, the SysteMoC language, the input for the proposed clustering-based design flow, is introduced and its MoC is classified in terms of the hierarchy of expressiveness as discussed in this chapter.
In this chapter, the first key contribution of this work, the SystemC Models of Computation (SysteMoC) modeling language [FHZT13*, FHT06*] for the Electronic System Level (ESL) will be presented. This language has strong formal underpinnings in data flow modeling with the distinction that the expressiveness of the data flow model used by an actor is not chosen in advance but determined from the implementation of the actor via classification [FHZT13*, ZFHT08*]. In order to determine the data flow model used by a SysteMoC actor by means of classification, the SysteMoC language, in contrast to SystemC, enforces a distinction between communication and computation of an actor. A further key point of the SysteMoC language is the ability of executable specifications modeled in SysteMoC to act as an input model for Design Space Exploration (DSE) as well as to serve as an evaluator [SFH+06*] during DSE to evaluate timing and power characteristics of the executable specification for the different design decisions that are to be explored. Note that parts of this chapter are derived from [FHT06*, FHT07*, ZFHT08*, FZK+11*, FHZT13*].

In the following, an outline for the rest of this chapter is given: In Section 3.1, the importance of SystemC for modeling at the ESL will be discussed. Then, the integration of an executable specification modeled in SysteMoC into the SystemCoDesigner [HFK+07*, KSS+09*] DSE methodology will be summarized in Section 3.2.

Next, the SysteMoC modeling language will be introduced, first by presenting the underlying Model of Computation (MoC) of SysteMoC in Section 3.3, and then by presenting the syntax of SysteMoC in Section 3.4. Subsequently, in Section 3.5, SysteMoC models are extended by a notion of hierarchy.

A data flow graph that is conforming to the MoC of SysteMoC can in theory also be extracted from a pure SystemC model if certain coding standards are enforced. In Section 3.6, an example of such an extraction will be given.

After a data flow graph conforming to the MoC of SysteMoC is available, a classification algorithm introduced in Section 3.7 is used to determine for each actor of the graph whether the actor belongs to one of the three static data flow models of computation, i.e., Homogeneous (Synchronous) Data Flow (HSDF), Synchronous Data Flow (SDF), and Cyclo-Static Data Flow (CSDF). The
classification algorithm provides only a sufficient criterion if a general SysteMoC actor conforms to one of the static data flow models of computation. This limitation stems from the fact that the problem in general is undecidable.

Finally, the related work will be discussed in Section 3.8.

3.1 Electronic System Level Design and SystemC

The research area called Electronic System Level (ESL) design tries to cope with the problem of rising hardware and software design complexity. Industry is increasingly adopting new design methodologies that employ abstraction levels beyond Register Transfer Level (RTL). The rising level of abstraction on the hardware design side gives rise to the opportunity to unify hardware and software modeling by a common modeling language. One of such approaches is the ESL ecosystem around the SystemC [GLMS02, Bai05] modeling language.

The SystemC framework is an open source C++ class library. It was initially promoted by the Open SystemC Initiative (OSCI) to facilitate an industry-wide standard for the interchange of high-level models. The Application Programming Interface (API) of this library has been standardized [Bai05] by the IEEE. This has enabled independent support of this API by various Electronic Design Automation (EDA) vendors like Mentor Graphics [Men13], Synopsys [Syn13], Imperas [Imp13], Forte Design Systems [For13], and many more.

Furthermore, existing high-level synthesis tools like Forte Cynthesizer [For13], Catapult C from Calypto Design Systems [Cal13], or Vivado from Xilinx [XIL14] allow the translation of behavioral SystemC models to the RTL level. Hence, these tools bridge the gap from a high-level behavioral implementation of an algorithm in SystemC to an implementation of the algorithm at RTL, which is required to realize the algorithm as an Application-Specific Integrated Circuit (ASIC) or in a Field Programmable Gate Array (FPGA). As SystemC itself is a C++-based class library, a transformation of SystemC into a pure C++ software implementation is also possible if the SystemC application is modeled at an appropriate level of abstraction and certain constraints are satisfied. Thus, SystemC is uniquely positioned as a design language for the ESL.

In [Tei12], Teich presents the so-called double roof model of hardware/software codesign. This model illustrates (cf. Figure 3.1) the usage of successive refinements in order to derive implementations containing hardware and software subsystems from a functional specification at ESL. The double roof model can be understood as an extension of the Y-chart approach [KDWV02] by explicitly separating the development chains for hardware and software.

In contrast to dedicated software languages like C, C++, and Java or dedicated hardware description languages like Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog Hardware Description
Figure 3.1: The *double roof* model of hardware/software codesign from [Tei12] shows the system level (shaded) at the top connecting the software (left) and hardware development chains (right) through successive synthesis refinements (vertical arrows). Each synthesis step maps a functional specification onto a structural implementation on the next lower level of abstraction.

Language (Verilog), the SystemC language has broad applicability to represent the functionality at various levels of abstraction depicted in the double roof model. SystemC was initially developed by OSCI to represent virtual prototypes and to serve as an input language for behavioral synthesis tools like Forte Cynthesizer [For13], Catapult C from Calypto Design Systems [Cal13], and Vivado from Xilinx [XIL14]. Virtual prototypes are situated in the double roof model [GHP+09] at the structural view of the system level. The behavioral view of the system level in the double roof model corresponds to the executable specification at ESL.

On the hardware development chain side of the double roof model, SystemC is used as an input and output language for behavioral synthesis tools. These behavioral synthesis tools translate an algorithmic problem description to the RTL level. This translation corresponds to a refinement at the architecture level of the double roof model from the behavioral view to the structural view. SystemC can also be used to represent the functionality at the logic level of abstraction. However, it is not well-suited for modeling at this level. Due to its heritage from C++, SystemC can also be used adequately to represent functionality at the module and block level of the software development chain side of the double roof model. In the following, SystemC is examined in more detail:
**SysteMoC**

- SystemC is a C++ class library. Thus, it can easily integrate models written in C, C++, or any language that supports C linkage.

- SystemC provides data types for hardware modeling, e.g., arbitrary length bit vectors, four valued logic types, corresponding arbitrary length logic vectors, arbitrary precision integers, fixed-point data types with various rounding and saturation modes, and many more. Furthermore, for modeling purposes the user can define his own data type by usage of common C++ constructs.

- At the core of a SystemC simulator resides a discrete event simulation kernel. Events can be generated and notified at user-specified times in the future. SystemC provides a mechanism to associate actions with an event. If the simulation time reaches the specified notification time of an event, the associated actions are executed. Based on the discrete event simulation kernel, SystemC provides a cooperative multitasking environment. The SystemC processes provided by the cooperative multitasking environment can be enabled by the notification of an event or suspend themselves waiting for the notification of an event.

- The structure of a SystemC design is borrowed from the world of hardware description languages. Hence, it corresponds to the design structure as seen in the VHDL and the Verilog. Therefore, SystemC designs represent the architecture of a system. The structure consists of concurrently executing modules. The modules communicate via ports. These ports should be the only way for modules to communicate with their environment. The channels implement a certain interface and are themselves connected to a port, which requires this interface. In the parlance of data flow, a module corresponds to an actor. However, SystemC channels are not restricted to First In First Out (FIFO) communication, but may implement any kind of communication fabric.

- Finally, SystemC provides a number of standard channels. These predefined channel types range from signals, as known from VHDL and Verilog, to registers, and FIFO channels.

The rising complexity of current designs stems not only from the increase in design functionality, but also from the potentially exponential increase in the number of ways this design functionality can be implemented in an embedded system. At the root of the double roof model (shaded in Figure 3.1) is a DSE

---

14 Due to the pervasiveness of C/C++-based software, every programming language is almost guaranteed to support C linkage.
3.2 Integration of SysteMoC and Design Space Exploration

A DSE methodology, e.g., like the ESL design flow (cf. Figure 3.2) used by SystemCoDesigner [HFK+07*, KSS+09*], typically starts with an architecture independent description of the functionality a system shall implement. This description, given by a so-called problem graph, encodes the functionality that will be implemented on a still to be determined architecture of the system. The data flow domain has proven to be an apt representation for the description of this functionality. In this case, the actors of the data flow graph represent the functionality. The channels connecting the actors specify the dependencies and the communication of the actors amongst each other. The possible variety of architectures is modeled by a so-called architecture graph. The architecture graph is composed of resources and edges connecting two resources that are able to communicate with each other. Finally, mapping edges encode for each actor of the problem graph the set of resources of the architecture graph onto which the actor can be bound. A mapping edge from an actor or channel of the problem graph to a resource of the architecture graph denotes that the actor or channel can be realized by an implementation via usage of this resource. The whole graph structure depicted in Figure 3.3 is called a specification graph [BTT98]. Next, the set of non-dominated individuals is explored by SystemCoDesigner using Multi-Objective Evolutionary Algorithms (MOEAs) [SD94, ZLT02] together with symbolic Boolean Satisfiability (SAT) based optimization techniques [HST06, LSG+09]. Note that each individual of the MOEA is given by an allocation of a certain number of vertices in the architecture graph and a feasible binding of vertices of the problem graph onto architecture graph vertices. In each iteration step in the DSE, the population of individuals is updated by the MOEA by means of crossover and mutation. In order to select parent individuals to perform crossover, the objectives for comparing individuals are estimated within the DSE.

To exemplify, consider the problem graph depicted in Figure 3.3. This problem graph corresponds to a data flow graph that implements Newton’s iterative algorithm for calculating the square root. The actor \( a_1 \) is the source produc-
Figure 3.2: Overview of an ESL design flow using the SYSTEMCODESIGNER DSE methodology.

ing the numbers for which the square root should be calculated. The actors $a_2$, $a_3$, and $a_4$ correspond to Newton’s iterative algorithm for calculating the square root of a number emitted by actor $a_1$. Finally, $a_5$ models the output of the calculated square root. The approximation step of the iterative square root calculation is realized by actor $a_3$, while checking the error bound in order to terminate the approximation is performed by actor $a_2$. Hence, actor $a_3$ requires floating point division, and actor $a_2$ requires floating point multiplication. Considering the architecture graph, two Central Processing Units (CPUs) $r_1$ and $r_2$, a dedicated hardware accelerator $r_3$, a memory $r_4$, and two buses $r_5$ and $r_6$ can be identified. While CPU $r_1$ possesses a hardware floating point unit, no such hardware support for floating point calculations is present in CPU $r_2$. Hence, floating point calculations must be emulated in software for CPU $r_2$. Thus,
while all actors can be mapped to both CPUs \( r_1 \) and \( r_2 \), the actors \( a_2 \) and \( a_3 \) will perform significantly worse on CPU \( r_2 \) than on CPU \( r_1 \). To improve this situation, the hardware accelerator \( r_3 \) for actor \( a_3 \) is connected to CPU \( r_2 \) via the point-to-point link \( r_6 \). The memory \( r_4 \), which is connected by the bus \( r_5 \) to the two CPUs \( r_1 \) and \( r_2 \), is used to provide the program memory required by the CPUs to implement the actors bound to these CPUs as well as the memory required to implement the channels \( c_1 - c_6 \). If the actor \( a_3 \) is bound to resource \( r_3 \), then the channels \( c_2 - c_3 \) are also bound to \( r_3 \), using the internal memory provided by this resource.

The problem graph of a specification graph can now be implemented in various ways (cf. Figure 3.4) by the provided architecture graph. Such an implementation is represented by a so-called \textit{implementation graph}. An implementation
3. SysteMoC

graph is a specification graph that has exactly one mapping edge activated for each actor and each channel. The activated mapping edge of an actor or channel decides onto which resource of the architecture graph the actor or channel is bound. Resources of the architecture graph not being a target of a mapping edge are eliminated. The DSE methodology tackles the problem of the exponential explosion of the number of ways the desired functionality can be implemented in an embedded system by performing an automatic optimization of the implementation of the functionality. The output of the automatic optimization is a set of non-dominated implementations of the functionality with respect to the objectives under evaluation by the DSE methodology (cf. Figure 3.2).

However, the optimization needs a way to compare the implementations with each other. Here, two kinds of evaluators are used: (1) analytical evaluators, e.g., the cost of the system, which is derived from adding all the costs of the resources used, and (2) simulative evaluators, e.g., the bandwidth of the implementation derived by a simulation of an executable specification for a given input stimulus. A simulative evaluator [SFH+06] is realized by parameterizing [SGHT10, XRW+12] a given executable specification, for example as part of SystemCoDesigner, in such a way that it conforms to the design decisions taken by the DSE for the implementation under evaluation. Hence, simulative evaluators can only be used if an executable specification of the system is available. Furthermore, the result of a simulative evaluator generally depends on the input stimulus given to the executable specification.

To exemplify, consider the four implementation graphs depicted in Figure 3.4. As can be seen, exactly one mapping edge is activated for each actor and each channel. The target of the mapping edge selects the resource onto which the actor or channel is bound. Each actor or channel will be realized by using the resource onto which it was bound to implement the functionality of the actor or to provide the storage for the FIFO channel. Shown are two implementations using a CPU without (cf. Figure 3.4a) and with (cf. Figure 3.4c) hardware floating point unit, an implementation (cf. Figure 3.4b) using a CPU without hardware floating point unit, but with an additional hardware accelerator that provides hardware support for the floating point division required by actor $a_3$, and an implementation (cf. Figure 3.4d) using both CPUs and the accelerator. The memory is always present to provide the storage for the FIFO channels and the program code for the actors. The buses are present as required in order to provide communication between the resources used by an implementation. It is evident that the non-functional design properties, e.g., the timing and hence the bandwidth, of an implementation will differ depending on the mapping of the actors and channels to the resources. Furthermore, the cost of an implementation will differ depending on the resources used by the implementation.

For DSE, SystemC may be a good choice when used as a front end language. However, some shortcomings of the language have to be removed. The under-
3.2 Integration of SysteMoC and Design Space Exploration

(a) Implementation graph of a solution using CPU $r_2$ requiring software emulation to support floating point operations

(b) Implementation graph of a solution where CPU $r_2$ is supported by an additional hardware accelerator $r_3$

(c) Implementation graph of a solution using CPU $r_1$ possessing a hardware floating point unit

(d) Implementation graph of a solution using all resources that have initially been specified in the architecture graph

Figure 3.4: Depicted above are four possible implementations of the desired functionality of the system.

lying problem for these shortcomings of SystemC is the focus of SystemC on architecture centric descriptions. Moreover, the following issues prevented the easy usage of general SystemC as a front end language for the SYSTEMCoDESIGNER DSE methodology:

- General SystemC models may implement any kind of communication fabric for the communication of SystemC modules amongst each other.
3. **SysteMoC**

- General SystemC models mix the functionality and the architecture used for implementing this functionality.

- General SystemC models do not separate computation and communication behavior of an actor.

The first issue can be solved by restricting SystemC applications to use only channel types exhibiting FIFO semantics like `sc_fifo`.

The second issue can be removed by insisting that the SystemC application is a purely functional model omitting annotations of timing values or any other non-functional properties like values for power consumption required to execute a part of the functionality. However, when insisting on this constraint, a SystemC application used as design entry becomes unusable for simulation-based evaluation during exploration. In contrast to the architecture independent and purely functional description that is used as a design entry, simulative evaluators require the presence of *architecture dependent non-functional properties* like power consumption values and execution times for all parts of the functionality in order to evaluate an implementation. The concrete values for these architecture dependent non-functional properties will, of course, be dependent on the design decisions taken by the DSE for the implementation under evaluation. Hence, the concrete values of these properties will vary amongst different simulations.

The third issue prevents the extraction of a certain MoC from each actor of a general SystemC application. This is important as the knowledge that a set of actors conforms to one of the static data flow models introduced in Section 2.2 enables the analysis of this set of actors. Examples of such analyses are deadlock detection or scheduling analysis information enabling the generation of more efficient schedules for these actors than would be possible without this information. With the algorithms presented in Sections 3.6 and 3.7, it is, however, possible to formulate sufficient conditions for a constrained subset of SystemC to conform to certain MoCs.\(^\text{15}\) Fortunately, these constraints turn out to be very similar to the synthesizable subset of SystemC.

In summary, general SystemC cannot be used as a front end language for DSE at the ESL, and the constrained subset as given above is unusable for simulation-based performance evaluation. Two approaches can be taken to resolve this problem. In the first approach, the SystemC model used as a design entry will be automatically annotated with the architecture dependent non-functional properties corresponding to the design decisions taken by the DSE, and then this annotated SystemC model must be compiled in order to be used as an evaluator by the DSE. In the second approach, the SystemC model used as a

\(^{15}\)Only a sufficient condition can be given as the determination of a MoC from a SystemC specification is undecidable in general.
design entry must support the ability to be run-time configurable based on the design decisions taken by the DSE for the implementation under evaluation with the varying concrete values for the architecture dependent non-functional properties. The first approach can produce simulators with higher performance than the second approach of run-time configurability. However, the first approach always incurs the overhead of compilation before the generated simulator can be executed. Thus, if the overall execution time required by a single simulation run is small when compared to the required time for compilation, as is typically the case, then the second approach will still be the faster one.

Consequently, SysteMoC—an extension of SystemC—uses the second approach of run-time configurability in order to tackle the problem. SysteMoC enables SystemC-based modeling to be used by SYSTEMCoDESIGNER as (1) a front end language, as (2) a simulative evaluator [SFH+06*][SGHT09, XRW+12] for the design decisions taken by the DSE for the implementation under evaluation, and as (3) a golden model from which virtual prototypes for selected implementations can be derived via usage of the synthesis back-end presented in Chapter 5. In particular, SysteMoC provides the following abilities on top of SystemC:

- SysteMoC provides the ability to automatically extract the problem graph from an executable specification modeled in this language.

- SysteMoC provides the ability to automatically extract a Finite State Machine (FSM) modeling the communication behavior of an actor for each actor of the problem graph.

- SysteMoC provides the ability to back annotate design decisions taken by the DSE via run-time configuration into the executable specification, e.g., timing and power behavior depending on the architecture chosen by the DSE.

The model extracted from the SysteMoC application via usage of the first two abilities will be presented in the next section. The missing information to complete the specification of the DSE problem, i.e., possible architectures and mappings, has to be added (cf. Figure 3.2) manually by the designer. The resulting specification graph can then be used by the SYSTEMCoDESIGNER DSE methodology to perform an automatic optimization of the implementation of the desired functionality encoded in the executable specification written in SysteMoC.
3. SysteMoC

3.3 The Model of Computation of SysteMoC

SysteMoC extends SystemC modules to communicate over particular \textit{FIFO channels} and enforces the usage of an FSM inside each module to express their communication behavior. Such an extended SystemC module will be called a SysteMoC actor. In actor-oriented design, \textit{actors} are objects which execute concurrently and can only communicate with each other via channels instead of method calls as known in object-oriented design. An actor \(a\) may only communicate with other actors through its dedicated actor input ports \(a.I\) and actor output ports \(a.O\).\(^{16}\) These ports are connected to channels. While SysteMoC supports various channel types, e.g., signals used to describe the synchronous reactive MoC, in this thesis the considered channels are constrained to have FIFO semantics. The basic entity of data transfer for this kind of channels is regulated by the notion of \textit{tokens} that are transmitted via these channels.

3.3.1 SysteMoC Actors

A SysteMoC actor consists of three parts:

- The actor \textit{input ports} and \textit{output ports} for communication.
- The actor \textit{functionality} for transforming tokens.
- The actor \textit{FSM} that encodes the \textit{communication behavior} of the actor.

These three parts are visualized in Figure 3.5 for the \texttt{SqrLoop} actor \(a_2\) from Figure 3.3.

As can be seen, a SysteMoC actor is very similar to a hierarchical Functions Driven by State Machines (FunState) model [TSZ\textsuperscript{+}99], e.g., as depicted in Figure 2.11 on page 26. The actor FSM corresponds to the FSM of the composite actor \(a_\gamma\). However, internal FIFOs and FSMs for the individual functions are discarded in SysteMoC actors. Moreover, the guards on the transitions of the actor FSM must also contain checks for the availability of space on the actor output ports as SysteMoC FIFOs are of limited capacity. Finally, in contrast to FunState, a transition of an actor FSM will not only trigger a function, called an \textit{action} in SysteMoC, but completes only after the action has finished execution. More formally, the following definition of a SysteMoC actor can be derived:

\textbf{Definition 3.1} (Actor [FHT06\textsuperscript{*}]). An actor is a tuple \(a = (P, F, R)\) containing a set of \textit{actor ports} \(P = I \cup O\) partitioned into \textit{actor input ports} \(I\) and \textit{actor output ports} \(O\), the \textit{actor functionality} \(F = F_{\text{action}} \cup F_{\text{guard}}\) partitioned into a set of actions and a set of guards, as well as the \textit{actor FSM} \(R\).

\(^{16}\)The ‘.\texttt{-}’-operator, e.g., \(a.I\), is used to denote member access of tuples whose members have been explicitly named in their definition, e.g., member \(I\) of the actor \(a\) from Definition 3.1.
3.3 The Model of Computation of SysteMoC

As can be seen, SysteMoC in contrast to Kahn Process Network (KPN) and FunState distinguishes its functions further into actions \( f_{\text{action}} \in \mathcal{F}_{\text{action}} \) and guards \( f_{\text{guard}} \in \mathcal{F}_{\text{guard}} \). To exemplify, the \( \text{SqrLoop} \) actor depicted in Figure 3.5 is considered. This actor has three actions \( \{ f_{\text{copyStore}}, f_{\text{copyInput}}, f_{\text{copyApprox}} \} = \mathcal{F}_{\text{action}} \) and one guard function \( \{ f_{\text{check}} \} = \mathcal{F}_{\text{guard}} \). Naturally, the guard \( f_{\text{check}} \) is only used in a transition guard of the FSM \( a_2.\mathcal{R} \) while the actions of the FSM are drawn from the set of actions \( \mathcal{F}_{\text{action}} \). Furthermore, there is a notion of a functionality state \( q_{\text{func}} \in Q_{\text{func}} \) of an actor. The functionality state is an abstract representation of the C++ variables in the real implementation of a SysteMoC actor. This functionality state is required for notational completeness, that is for the formal definitions of action and guard functions.\(^{17}\)

\[ f_{\text{action}} : Q_{\text{func}} \times S_{I} \rightarrow Q_{\text{func}} \times S_{O} \]

\[ f_{\text{guard}} : Q_{\text{func}} \times S_{I} \rightarrow \{ t, f \} \] \(^{18}\)

Both types of functions depend on the functionality state of their actor. Furthermore, an action may update this state, while a guard function may not. Hence, all SysteMoC actor firings are sequentialized over this functionality state, and therefore multiple actor firings of the same actor cannot be executed in parallel. That is, in data flow parlance, all SysteMoC actors have a virtual self-loop with

\(^{17}\)In the definition below, the customary notation \( |X| \) is used to denote the cardinality of a set \( X \).  
\(^{18}\)In the following, \( t \) and \( f \) are used to denote Boolean truth values.
one initial token that corresponds to \( q_{\text{func}} \in Q_{\text{func}} \). However, as the functionality state is not used for any optimization and analysis algorithms presented in this dissertation, it has been excluded from the definition of an actor. Hence, for analysis purposes the evaluation of a guard function is assumed to always be able to return both \( t \) (true) and \( f \) (false).

Furthermore, the \texttt{cons} and \texttt{prod} notation of FunState is reused to specify the length of token sequences that may be accessed by the guard functions \( f_{\text{guard}} \in \mathcal{F}_{\text{guard}} \) as well as accessed and written by the action functions \( f_{\text{action}} \in \mathcal{F}_{\text{action}} \).

- The \texttt{cons} : \((I \times \mathcal{F}) \rightarrow \mathbb{N}\) function specifies for each input port/channel the number of tokens required on this channel to compute the action or guard \( f \in \mathcal{F} \).\(^{19}\) Note that, in contrast to actions, the computation of a guard function does not consume any tokens required for its evaluation.

- The \texttt{prod} : \((\mathcal{F}_{\text{action}} \times \mathcal{O}) \rightarrow \mathbb{N}\) function specifies for each output port/channel the number of tokens produced by the action \( f_{\text{action}} \in \mathcal{F}_{\text{action}} \) for this channel.

An actor FSM resembles the FSM definition from FunState, but with slightly different definitions for actions and guards. More formally, the definition is given below:

**Definition 3.2** (Actor FSM [FHT06*]). The FSM \( \mathcal{R} \) of an actor \( \alpha \) is a tuple \((Q, q_0, T)\) containing a finite set of states \( Q \), an initial state \( q_0 \in Q \), and a finite set of transitions \( T \). A transition \( t \in T \) itself is a tuple \((q_{\text{src}}, k, f_{\text{action}}, q_{\text{dst}})\) containing the source state \( q_{\text{src}} \in Q \), from where the transition is originating, and the destination state \( q_{\text{dst}} \in Q \), which will become the next current state after the execution of the transition starting from the current state \( q_{\text{src}} \). Furthermore, if the transition \( t \) is taken, then an action \( f_{\text{action}} \) from the set of functions of the \textit{actor functionality} \( \alpha.\mathcal{F}_{\text{action}} \) will be executed. Finally, the execution of the transition \( t \) itself is guarded by the \textit{guard} \( k \).

A transition \( t \) will be called an \textit{outgoing transition} of a state \( q \) if and only if the state \( q \) is the source state \( t.q_{\text{src}} \) of the transition. Correspondingly, a transition will be called an \textit{incoming transition} of a state \( q \) if and only if the state \( q \) is the destination state \( t.q_{\text{dst}} \) of the transition. Furthermore, a transition is \textit{enabled} if and only if its guard \( k \) evaluates to \( t \) and it is an outgoing transition of the current state of the actor.

An actor is enabled if and only if it has at least one enabled transition. The firing of an actor corresponds to a non-deterministic selection and execution of

\(^{19}\)Remember that in this thesis, the notion of ports are used interchangeably with the channels connected to these ports. Thus, both \texttt{cons} as well as \texttt{prod} can also be used with channels instead of ports.
3.3 The Model of Computation of SysteMoC

one transition out of the set of enabled transitions of the actor. In general, if multiple actors have enabled transitions, then the transition, and hence its corresponding actor, is chosen non-deterministically by the SysteMoC run-time system out of the set of all enabled transitions in all actors. In summary, the execution of a SysteMoC model can be divided into three phases:

- Determine the set of enabled transitions by checking each transition in each actor of the model. If this set is empty, then the simulation of the model will terminate.
- Select a transition \( t \) from the set of enabled transitions, and fire the corresponding actor by executing the selected transition \( t \), thus computing the associated action \( f_{\text{action}} \).
- Consume and produce tokens as encoded in the selected transition \( t \). This might enable new transitions. Go back to the first step.

In contrast to FunState, a guard \( k \) is more structured. Moreover, it is partitioned into the following three components:

- The input guard which encodes a conjunction of input predicates on the number of available tokens on the input ports, e.g., \( #i_1 \geq 1 \) denotes an input predicate that tests if at least one token is available at the actor input port \( i_1 \).
- The output guard which encodes a conjunction of output predicates on the number of free places on the output ports, e.g., \( #o_1 \geq 1 \) denotes an output predicate that tests if at least one free place is available at the actor output port \( o_1 \).
- The functionality guard which encodes a logical composition of guard functions of the actor, e.g., \( \neg f_{\text{check}} \). Hence, the functionality guard depends on the functionality state and the token values on the input ports.

Therefore, the guard \( k \) is encoded by a Boolean function depending on the number of available tokens (\( \#i_1, \#i_2, \ldots, \#i_{|I|} \)) in \( \mathbb{N}_{0}^{|I|} \) and the number of free places (\( \#o_1, \#o_2, \ldots, \#o_{|O|} \)) in \( \mathbb{N}_{0}^{|O|} \) on the actor input and output ports, respectively. Furthermore, due to the guard function \( f_{\text{guard}} \in \mathcal{F}_{\text{guard}} \) that can be used inside a guard \( k \), a guard typically requires the actor functionality state \( q_{\text{func}} \in \mathcal{Q}_{\text{func}} \) and the token sequences \( (s_1, s_2, \ldots, s_{|I|}) \in S^{[I]} \) available on the input ports \( I \) in order to determine whether it is enabled or not.

\[
k : \mathbb{N}_{0}^{|I|} \times \mathbb{N}_{0}^{|O|} \times \mathcal{Q}_{\text{func}} \times S^{[I]} \to \{ t, f \}
\]
3. SysteMoC

Input and output predicates are only allowed to be composed via logical conjunction. Thus, there exists a unique least vector of available tokens and free places \( \mathbf{n} = (n_{i_1}, n_{i_2}, \ldots, n_{i_{|I|}}, n_{o_1}, n_{o_2}, \ldots, n_{o_{|O|}}) \) that are required to enable the guard \( k \). More formally, each transition \( t \in T \) satisfies the below given condition defining its unique least vector \( \mathbf{n} \).

\[
\exists \mathbf{n} \in \mathbb{N}_0^{|P|} : \forall s \in S^{|I|}, q_{\text{func}} \in Q_{\text{func}} : k(\mathbf{n}, q_{\text{func}}, s) \implies \forall \mathbf{n}' \not\geq \mathbf{n} : -k(\mathbf{n}', q_{\text{func}}, s)
\]

Furthermore, the \text{cons} and \text{prod} notation will be used to refer to the entries of the above specified unique least vector \( \mathbf{n} \).

- The \text{cons} : \((T \times I) \to \mathbb{N}\) function specifies for each transition and input port/channel combination the number of tokens \( n_i = \text{cons}(t, i) \) consumed if the transition is taken.

- The \text{prod} : \((T \times O) \to \mathbb{N}\) function specifies for each transition and output port/channel combination the number of tokens \( n_o = \text{prod}(t, o) \) produced if the transition is taken.

In other words, this unique least vector is required to enable the transition \( t \). Note that it is also the number of tokens that are consumed/produced on the input/output ports if the transition is taken. Hence, for a SysteMoC model to be well formed, the number of tokens accessed on the different input and output ports by an action \( t.f_{\text{action}} \) associated with the transition \( t \) as well as the guard functions \( f_{\text{guard}} \) used in the transition guard \( t.k \) has to conform to the consumption and production rates of the transition. That is, \( \forall i \in I : \text{cons}(i, t.f_{\text{action}}) = \text{cons}(t, i) \land \forall f_{\text{guard}} \) contained in \( t.k : \text{cons}(i, f_{\text{guard}}) \leq \text{cons}(t, i) \) and \( \forall o \in O : \text{prod}(t.f_{\text{action}}, o) = \text{prod}(t, o) \).

3.3.2 SysteMoC Network Graphs

In contrast to the data flow models introduced in Chapter 2, FIFO channels of limited capacity are used within SysteMoC models. Note that the determination of limited channel capacities that do not introduce deadlocks into a model is in general undecidable [Par95] for dynamic data flow graphs. The introduction of limited channel capacities into a SysteMoC model is necessary, however, when refining a model for subsequent implementation in hardware or software. Here, synthesis back-ends only support limited channels.

Actor-oriented designs are often represented by bipartite graphs consisting of channels \( c \in C \) and actors \( a \in A \), which are connected via point-to-point connections from an actor output port \( o \) to a channel and from a channel to an actor input port \( i \). In the following, such a representation is called a network graph, an example of which can be seen in Figure 3.6. This graph corresponds
3.3 The Model of Computation of SysteMoC

Figure 3.6: The network graph $g_{\text{sqr, flat}}$ displayed above implements Newton’s iterative algorithm for calculating the square roots of an infinite input sequence generated by the $\text{Src}$ actor $a_1$. The square root values are generated by Newton’s iterative algorithm $\text{SqrLoop}$ actor $a_2$ for the error bound checking and $a_3 - a_4$ to perform an approximation step. After satisfying the error bound, the result is transported to the $\text{Sink}$ actor $a_5$.

exactly to the problem graph contained in the specification graph depicted in Figure 3.3.

In fact, in the SystemCoDesigner DSE methodology, the names problem graph and network graph denote the same graph structure. However, for historical reasons, the name problem graph [BTT98] is commonly used if this graph is considered from the DSE perspective. Furthermore, from a broader DSE perspective, a problem graph may not necessarily exhibit data flow semantics and only encode some kind of data communication. In contrast to this, the name network graph is used if data flow aspects are in focus. More formally, the following definition of a non-hierarchical network graph can be derived:

**Definition 3.3** (Non-Hierarchical Network Graph [FHT06*, FHZT13*]). A non-hierarchical network graph is a directed bipartite graph $g_n = (V, E)$ containing a set of vertices $V = C \cup A$ partitioned into channels $C$ and actors $A$, a set of directed edges $e = (v_{\text{src}}, v_{\text{snk}}) \in E \subseteq (C \times A.I) \cup (A.O \times C)$ from channels
\[ c \in C \text{ to actor input ports } i \in A.I \text{ as well as from actor output ports } o \in A.O \text{ to channels.}\]  

Finally, the delay function \( \text{delay} \) (reused from Definition 2.1) and the channel capacity function \( \text{size} \) are used to denote a finite sequence of initial tokens as well as the channel capacity denoting the maximal number of tokens a channel can store, respectively.

\[
\text{delay} : C \to \mathbb{V}^* \\
\text{size} : C \to \mathbb{N}_0
\]

3.3.3 SysteMoC FIFO Channels

It can be seen (cf. Figure 3.7) that the actor FSM controls which action to execute, decides how many tokens to consume and produce on each port by advancing the read or write pointer of the corresponding FIFO, and controls the size of a random access region (the two big black boxes in Figure 3.7) of each FIFO. Hence, as only tokens inside the random access region can be modified by the actions or read by the guards, the actor FSM controls the communication behavior of the actor. A random access region has been introduced here, to satisfy the need of general SysteMoC applications for random access and for multiple access to the same token value. For example, in all action and guard functions, the syntax \( \text{portvariable}[<n>] \) is used for accessing the \( n \)th token in the random access region. This region is relative to the read or write pointer of the connected FIFO. An implementation based on conventional FIFOs and internal buffering of the tokens inside the actor itself, would have been also feasible, but would mix implementation decisions, i.e., using internal buffering, with model requirements.

3.4 The SysteMoC Language

In this section, the SysteMoC language, a class library based on SystemC, and its syntax will be presented. Newton’s iterative square root algorithm (cf. Figure 3.6) will be used as a running example to give a short introduction into the syntax of the SysteMoC language.

\[20\] The ‘.’-operator has a trivial extension to sets of tuples, e.g., \( A.I = \bigcup_{a \in A} a.I \), which is used throughout this document.
3.4 The SysteMoC Language

![Diagram of SysteMoC FIFO and source-sink example]

**Figure 3.7:** A simple source-sink example to explain the semantics of a SysteMoC FIFO associated with a channel

### 3.4.1 Specification of Actors

Each actor in SysteMoC is represented by a C++ class, e.g., the class `SqrLoop` as defined in the following Example 3.1 is representing the actor $a_2$ shown in Figure 3.6.

**Example 3.1 SysteMoC implementation of the SqrLoop actor $a_2$**

```cpp
class SqrLoop: public smoc_actor {
public:
    smoc_port_in<double> i1, i2;
    smoc_port_out<double> o1, o2;
private:
    double v;

    void copyStore() {o1[0] = v = i1[0];}
    void copyInput() {o1[0] = v;}
    void copyApprox() {o2[0] = i2[0];}
    bool check() const {return fabs(v-i2[0]*i2[0])<0.01;}

    smoc_firing_state start, loop;
public:
    SqrLoop(sc_module_name name);
};
```

Each SysteMoC actor is derived from the `smoc_actor` base class (cf. Example 3.1 Line 1) provided by the SysteMoC library. The input and output
ports of an actor are specified by member variables of type \texttt{smoc\_port\_in} and \texttt{smoc\_port\_out} as exemplified in Example 3.1 Lines 3 and 4, respectively.\footnote{Standard SystemC FIFO ports could not be used as the semantics of SysteMoC FIFOs extends standard FIFO semantics by the random access region defined in Section 3.3.3.} Furthermore, actors can have member variables, e.g., the variable \(v\) declared in Line 6, that realize the functionality state \(q_{\text{func}} \in Q_{\text{func}}\) defined in Definition 3.1. The actions and guards of an actor are represented by methods of the class, e.g., the methods \texttt{copyStore}, \texttt{copyInput} and \texttt{copyApprox} (cf. Lines 8 to 10) correspond to the actions \(f_{\text{copyStore}}, f_{\text{copyInput}},\) and \(f_{\text{copyApprox}}\) of the \texttt{SqrLoop} actor, while the method \texttt{check} represents the guard function \(f_{\text{check}}\). The \texttt{copyStore} method is responsible for forwarding the input token value on input port \(i_1\) to the output port \(o_1\) and storing the value into the functionality state represented by the member variable \(v\) for later replication on the output port \(o_1\) by the method \texttt{copyInput}. This replication is required if the achieved accuracy by one square root iteration step by actor \(a_3\) is below the bound determined by the guard \texttt{check}. On the other hand, if the approximation is within the error bound, then the action \texttt{copyApprox} is executed to forward this approximation to the output port \(o_2\).

Note that actions themselves do not control token production or consumption, but only read the values of input tokens for data processing and generation of values of output tokens. The communication behavior, i.e., token production and consumption, is controlled solely by the actor FSM (cf. Definition 3.2) as will be detailed next.

### 3.4.2 Specification of the Communication Behavior

The communication behavior of an actor is an abstraction from the functional behavior of the actor. The abstraction of the communication behavior concentrates on the tokens which are consumed and produced by an actor. In SysteMoC, the abstraction is done by representation of the communication behavior by an FSM (cf. Definition 3.2). The actions and guards of the actor functionality can only read and write values of tokens contained in the random access region defined in Section 3.3.3. This random access region is controlled by the actor FSM in order to enforce the specified communication behavior on the actor functionality. Note that if no interaction between the actor FSM and the functionality state \(q_{\text{func}}\) of an actor would be permissible, then SysteMoC actors could not implement general Kahn actors. Hence, SysteMoC uses the notion of guard functions to implement this interaction. To exemplify, the actor FSM of the \texttt{SqrLoop} actor is depicted in Figure 3.8. This FSM is constructed by the code shown in Example 3.2 Lines 10 to 23.
### 3.4 The SysteMoC Language

![Figure 3.8: The actor FSM of the SqrLoop actor $a_2$](image)

Example 3.2 SysteMoC implementation of the SqrLoop actor $a_2$

```cpp
class SqrLoop : public smoc_actor {
    ...
    smoc_firing_state start, loop;
public:
    SqrLoop(sc_module_name name)
        : smoc_actor (name, start),
        i1("i1"), i2("i2"), o1("o1"), o2("o2"),
        start("start"), loop("loop")
    {
        start =
            i1(1) >>
            o1(1) >>
            CALL(SqrLoop::copyStore) >> loop
    ;
    loop =
        (i2(1) && !GUARD(SqrLoop::check)) >>
        o1(1) >>
        CALL(SqrLoop::copyInput) >> loop
    |
        (i2(1) && GUARD(SqrLoop::check)) >>
        o2(1) >>
        CALL(SqrLoop::copyApprox) >> start
    ;
    }
};
```

The states $Q$ of the actor FSM themselves are represented by member variables of type `smoc_firing_state`, e.g., the state `start` and `loop` in Line 3, that is $Q = \{ q_{start}, q_{loop} \}$. The initial state of the actor FSM is determined by
providing the base class smoc_actor with the corresponding state. For the SqrLoop actor, the initial state $q_0$ is set by Line 6 to $q_{\text{start}} (\text{start})$.

The syntax for naming SystemC and also SysteMoC entities is demonstrated in Lines 7 and 8, where the actor input and output ports as well as the states of the actor FSM are named, respectively. However, in the interest of conciseness, it will be assumed that all SystemC/SysteMoC entities are named like their declarations in the source code shown in the following examples, but this naming will not be shown explicitly.

The transition $t_1$ from $q_{\text{start}}$ to $q_{\text{loop}}$ is given in Lines 11 to 13. Transition $t_2$ and $t_3$ are defined accordingly in Lines 16 to 18 and Lines 20 to 22. As can be seen from Example 3.2, a transition has the following syntax: $(\text{input guard } \&\& \text{ functionality guard}) \Rightarrow \text{output guard} \Rightarrow \text{CALL( action function )} \Rightarrow \text{destination state}$. Hence, a transition in SysteMoC syntax is divided into four components joined via the ‘$\Rightarrow$’-operator. To be more precise, only the last part, the destination state, is mandatory, while the first three components are optional. For example, if no outputs are produced, then the ‘output guard’ part can be dropped from the transition definition. Likewise, if the transition does not depend on the functionality state, then the ‘$\&\& \text{ functionality guard}$’ part can be dropped from the transition definition. In the following, the parts of a transition definition are considered in more detail:

- The input guard is encoded by a logical conjunction of input predicates on the number of available tokens on the input ports. The notation of an input predicate $\#i \geq n$ denoting the test for at least $n$ available tokens on input port $i$ is reflected in SysteMoC syntax by the expression `input_port_variable(n)`, e.g., the first part of transition $t_1$ from Line 11 consists only of an input guard with the single input predicate $i1(1)$, which denotes that at least one token on input port $i1$ is needed. Input predicates can only be composed by a conjunction that is represented by the C++ logical and operator ‘$\&\&$’ in SysteMoC syntax, e.g., $(i1(1) \&\& i2(1))$ is permissible while $(i1(1) || i2(1))$ or any other composition is forbidden.

- The functionality guard is encoded by a logical composition of guard functions of the actor. The guard functions are given via the GUARD syntax.

---

22 The need to name entities stems from the fact that SystemC, and therefore also SysteMoC, is a C++ class library based approach. The inherent deficiencies [GD09] of C++ with regard to program code introspection prevent the SystemC run-time system from acquiring the names under which the SystemC and SysteMoC entities are defined in the source code. Furthermore, without proper names for the states, ports, channels, actors, etc., the SystemC/SysteMoC run-time system cannot generate proper diagnostics messages. Hence, almost every SystemC/SysteMoC entity can be named by providing a string as its first parameter.
The mathematical definition of a guard function $f_{\text{guard}} : Q_{\text{func}} \times S[\mathcal{I}] \rightarrow \{t, f\}$ does not return a functionality state. To enforce this property, it is required that guard functions are implemented by \texttt{const} member functions of the actor. Example 3.1 Line 11 demonstrate this declaration of guard functions. The C++ compiler enforces that \texttt{const} member functions, and hence guard functions, can not modify the actor variables or call other non-\texttt{const} member functions that might modify those variables. Guard functions can, by definition, be composed via logical composition. Therefore, SysteMoC allows the composition of guard functions via the C++ ‘!’, ‘|’, ‘&&’, ‘==’, and ‘!’ operators.

If both \textit{input guard} and \textit{functionality guard} are present, then they are joined by a conjunction, e.g., Lines 16 and 20 of Example 3.2 which combine the input predicate $i2(1)$ with the negated guard function $\neg f_{\text{check}}$ and the guard function $f_{\text{check}}$, respectively.

- The \textit{output guard} is encoded by a logical conjunction of output predicates on the number of free places required on the output ports. The SysteMoC syntax for \textit{output guards} is analogous to the syntax for \textit{input guards}. The Lines 12 and 17 of Example 3.2 are demonstrating the usage of an output guard that consists of the single output predicate $o1(1)$ denoting a request for at least one free place on the channel connected to the output port $o1$, that is $\#o_1 \geq 1$.

- The \textit{action function} is given via the \texttt{CALL} syntax. To exemplify, the actions of the transitions $t_1$, $t_2$, and $t_3$ are specified, respectively, in Lines 13, 18, and 22 to call the three member functions $\text{copyStore}$, $\text{copyInput}$, and $\text{copyApprox}$. These three member functions correspond to the \textit{action functions} $f_{\text{copyStore}}$, $f_{\text{copyInput}}$, and $f_{\text{copyApprox}}$ of the mathematical model.

- Finally, the \textit{destination state} of the transition is specified last. In contrast to the previous four parts, which are optional, the destination state is a mandatory part of each transition specification.

In SysteMoC, each FSM state is defined explicitly by an assignment of a list of outgoing transitions. To exemplify, the state $q_{\text{start}}$ is defined in Example 3.2 Line 10 by assigning transition $t_1$ (Lines 11 to 13) to it. If a state has multiple outgoing transitions, e.g., like state $q_{\text{loop}}$ with the transitions $t_2$ (Lines 16 to 18) and $t_3$ (Lines 20 to 22), then the outgoing transitions are joined via the ‘|’-operator (Line 19) and assigned to the state variable (Line 15).
3.4.3 Specification of the Network Graph

Similar to actors, network graphs are represented as C++ classes. All actors and channels of a network graph must be instantiated in the constructor of the class representing the network graph. To exemplify, the SqrRoot class listed in Example 3.3 corresponding to the network graph shown in Figure 3.6 is considered. As can be seen in Line 1, the class representing the network graph is derived from the smoc_graph base class provided by the SysteMoC library. The actors $a_1, a_2, \ldots, a_5$ of the network graph are declared in the Lines 3 to 7 and instantiated in the constructor in Line 11. Finally, the FIFO channels $c_1, c_2, \ldots, c_6$ are specified in the Lines 12 to 20. The SqrRoot example uses the connectNodePorts method of SysteMoC to connect input to output ports. In the simplest case (Lines 12 to 15), the connectNodePorts methods takes two arguments the output port $o$, from where the channel starts, and the input port $i$, which is the destination of the channel.

Example 3.3 Specification of the network graph for the SqrRoot algorithm

```cpp
class SqrRoot : public smoc_graph {
    protected:
        Src a1;
        SqrLoop a2;
        Approx a3;
        Dup a4;
        Sink a5;
    public:
        SqrRoot(sc_module_name name)
            : smoc_graph(name),
            a1("a1"), a2("a2"), a3("a3"), a4("a4"), a5("a5"){
            connectNodePorts(a1.o1, a2.i1); // c1
            connectNodePorts(a2.o1, a3.i1); // c2
            connectNodePorts(a2.o2, a5.i1); // c6
            connectNodePorts(a4.o2, a2.i2); // c5
            connectNodePorts(a3.o1, a4.i1, // c4
                smoc_fifo<double>(2));       // size(c4) = 2
            connectNodePorts(a4.o1, a3.i2, // c3
                smoc_fifo<double>(3)        // size(c3) = 3
                << 2);                     // delay(c3) = (2)
        }
};
```
However, it is also possible to explicitly parameterize the created channel \( c \) with its channel capacity of \( \text{size}(c) \) tokens and a sequence of initial tokens \( \text{delay}(c) \). To exemplify, consider Lines 17 and 19, where a third parameter, the \textit{channel initializer} \texttt{smoc_fifo<T>(n) « initial tokens}..., is given to the method \texttt{connectNodePorts}. If no channel initializer is given, then a FIFO channel with a channel capacity of one token and without any initial tokens is created between the output port \( o \) and the input port \( i \). If a channel initializer is given, then it must be parameterized with the data type \( T \) carried by the channel, the channel capacity \( n \) in units of tokens, and a possibly empty sequence of initial tokens provided via usage of the '«'-operator. To exemplify, consider Line 17 of Example 3.3, where a FIFO channel with a channel capacity of two tokens is created between the ports \( a_3.o_1 \) and \( a_4.i_1 \). In this case, the channel initializer is parameterized with the C++ \texttt{double} data type, denoting that the token values carried by the channel will be of the C++ \texttt{double} data type. An example for providing an initial token is given in Line 20, where the '«'-operator is used to provide the \texttt{double} value 2 as an initial token for the created channel between \( a_4.o_1 \) and \( a_3.i_2 \).

### 3.5 Hierarchical SysteMoC Models

For the specification and modeling of complex systems, a notion of hierarchy is very helpful. Furthermore, hierarchy is used in Chapter 4 to describe algorithms for model refinements of SysteMoC models. In particular, hierarchy will be used to group an \textit{island of static actors} into a \textit{cluster} for later refinement by the \textit{clustering methodology} in order to optimize the generated schedule for the static actors contained in the island in such a way that the performance of the whole graph in terms of latency and throughput is optimized. Hence, hierarchy will be introduced into SysteMoC models by allowing a hierarchical specification of both network graphs and FSMs.

#### 3.5.1 Hierarchical Finite State Machines

While FSM is a formal model for reasoning about control flow, the modeling of complex control flow via non-hierarchical FSM specifications has turned out to be inadequate [Har87]. Hence, SysteMoC supports hierarchical FSM specifications [ZH+10*] similar to the statecharts [Har87] notion as introduced by Harel. In the following, the semantics of these hierarchical FSM specifications will be explained and the choice for \textit{interleaving} semantics instead of \textit{concurrency} for these hierarchical FSM specifications will be motivated. The notation of hierarchical FSM specifications allows to concisely specify complex control flow. The two main features responsible for this advantage are as follows:
3. SysteMoC

- Introduction of hierarchy by using so-called xor states (cf. Figure 3.9(a)). This allows the grouping of states implementing a certain functionality into one xor state, e.g., usually used to specify subsystems in a more complex specification. Furthermore, this enables one specified transition to represent an outgoing transition for each state contained inside an xor state, e.g., some form of reset transition to abort the execution of a subsystem or switch modes from one subsystem to another. The name xor state is derived from the invariant that if the FSM is in an xor state, then it is in exactly one of the states contained in the xor state.

- Representation of concurrency/interleaving by using so-called and states (cf. Figure 3.10). In the non-hierarchical case, the modeling of the control flow of multiple concurrent or interleaved executions will require a form of product FSM, and hence the number of states needed to be created by hand will exhibit an exponential explosion. In contrast to this, and states do create these product FSMs implicitly and no longer require hand modeling. Hence, the modeling of much larger specifications becomes feasible in a compact way.

![Diagram of xor and and states](image)

**Figure 3.9:** Example of expressing hierarchy in SysteMoC FSMs via usage of xor states

To exemplify xor states, consider the hierarchical FSM depicted in Figure 3.9(a). It consists of one xor state $q_D$ containing the two leaf states $q_A$ and $q_C$ as well as a leaf state $q_B$ on the same hierarchy level as $q_D$. The corresponding non-hierarchical FSM is depicted in Figure 3.9(b). Transitions between leaf states have the same semantics as in the non-hierarchical FSM model, e.g., the transition...
3.5 Hierarchical SysteMoC Models

from \( q_A \) to \( q_C \). These transitions may even connect leaf states inside different xor states, e.g., the transition from \( q_B \) to \( q_C \). Transitions with an xor state as source state are attached as outgoing transition to each state contained in the xor state, e.g., the transition from \( q_B \) to \( q_C \). Transitions with an xor state as destination state will be mapped in the non-hierarchical FSM to a transition where the destination state is determined by the initial state of the target xor state, e.g., the transition from \( q_B \) to \( q_D \) corresponds to the transition from \( q_B \) to \( q_A \) in Figure 3.9b.

![Figure 3.10: Example of a hierarchical FSM with an and state](image)

To exemplify and states, consider the (hierarchical) statechart FSM depicted in Figure 3.10. One central decision to determine the semantics of and states is the question if and states model concurrency or interleaving. As the depicted FSM belongs to the synchronous reactive domain [BCE+03], it has input events \((\alpha, \beta, \gamma)\) instead of input predicates and emits output events \((\text{emit}(\xi))\) instead of output predicates. In the synchronous reactive domain, transitions are assumed to be taken instantaneously\(^{23}\) and events may vanish between successive transitions. Hence, to prevent events to be lost, an and state models concurrency in the synchronous reactive domain, that is all transitions enabled by the set of input events are taken concurrently. Therefore, in statecharts, a firing of the hierarchical FSM can change the state in multiple xor states contained in an and state, e.g., as can be seen in the four transitions from \( q_{B,C} \) to \( q_{C,F} \) in Figure 3.11a. The actions of these four transitions are derived by parallel composition (as denoted by the \( \| \) operator) from the actions \( f_1, f_2, f_3 \), and \( f_4 \) in the individual xor states \( q_A \) and \( q_D \) contained in the and state.

\(^{23}\)In practice, this means faster than the worst case inter arrival time of events from the environment of the synchronous reactive system.
However, the concurrency interpretation has the following implications:

- The parallel composition of two actions, e.g., \( f_1 \parallel f_2 \), must be a well defined operation. However, when the actions \( f_1, f_2 : Q_{\text{func}} \rightarrow Q_{\text{func}} \) manipulate a shared functionality state \( q_{\text{func}} \in Q_{\text{func}} \), then this is not generally the case. For parallel composition to be a well defined operation, the next functionality state resulting from applying both \( f_1 \) and \( f_2 \) must be independent of the sequence of application of the actions to the current functionality state, that is \( f_1 \circ f_2 \equiv f_2 \circ f_1 \). Note that in general, actions encoded in SysteMoC do not have this property.

- It must be possible to create a union of output events generated by the set of enabled transitions. This is trivially possible if transitions generate events, then the union corresponds to the set union. To exemplify, consider the transitions with the actions \( f_1 \) and \( f_3 \) from Figure 3.10. Both transitions generate the output event \( \xi \), thus the transition with the parallel composition \( f_1 \parallel f_3 \) from Figure 3.11a will also only generate \( \xi \). However,

\[ f \circ g(x) \equiv f(g(x)) \]

As defined above, actions not only manipulate the functionality state, but are also given the values of the input tokens and calculate the values of the output tokens. However, the problem described here is independent of this fact and only based on the manipulation of the functionality state.

\[ \circ \]

The '\( \circ \)'-operator, e.g., \( f \circ g \), is used to denote function composition.
in SysteMoC, the output event $\xi$ will correspond to an output predicate on the same output port $o$. Thus, two token values will be generated by $f_1$ and $f_3$ for one output token.

Due to the above given implications, interleaving semantics has been chosen for hierarchical SysteMoC FSMs. Hence, actions of taken transitions are executed sequentially, while concurrency is provided due to the data flow nature of SysteMoC.

In general, a hierarchical FSM can be constructed by an arbitrary nesting of xor states and and states inside each other. This nesting is terminated by leaf states which do not contain further states. In SysteMoC, xor states and and states are represented by the C++ classes smoc_xor_state and smoc_and_state, respectively. Leaf states are represented by the C++ class smoc_firing_state, as known from the specification of non-hierarchical FSMs in SysteMoC.

### 3.5.2 Hierarchical Network Graphs

Next, the notion of clusters will be introduced. Clusters are used to introduce hierarchy into the non-hierarchical network graph model. A cluster is basically a data flow graph $g$ with input and output ports. Such a graph will be called an open system as it exchanges data with its cluster environment via its cluster input and output ports. Furthermore, the availability of tokens on the input ports of the cluster will be controlled from outside the cluster by the cluster environment. This is in contrast to the closed system nature of the traditional definition of a data flow graph as given in Definition 2.1. Traditional data flow graphs might communicate with an outside environment by using dedicated source and sink actors representing interfaces to this outside environment. However, the consumption and production of tokens by these dedicated source and sink actors is controlled by the scheduler of the data flow graph itself and not by the environment.

Hierarchy is introduced into a cluster or network graph by allowing it to contain subclusters, i.e., the vertices $g.V$ of a cluster are partitioned into actors $a \in g.A$, channels $c \in g.C$, and additionally also subclusters $g_\gamma \in g.G$. More formally, a cluster can be defined as follows:

**Definition 3.4 (Cluster [FHZE13*])**. A cluster is a directed bipartite graph $g = (P, V, E)$ containing a set of cluster ports $P = I \cup O$ partitioned into cluster input ports $I$ and cluster output ports $O$, a set of vertices $V = C \cup A \cup G$ partitioned into channels $C$, actors $A$, and subclusters $G$, a set of directed edges $e = (v_{src}, v_{snk}) \in E \subseteq ((C \cup I) \times (A.I \cup G.I)) \cup ((A.O \cup G.O) \times (C \cup O))$ from channels $c \in C$ or cluster input ports $i' \in I$ to actor or subcluster input ports $i \in A.I \cup G.I$ as well as from actor or subcluster output ports $o \in A.O \cup G.O$ to
channels or cluster output ports $o' \in G.O$. These edges are further constrained such that exactly one edge is incident to each actor, subcluster, or cluster port and the in-degree and out-degree of each channel in the graph is exactly one. Moreover, the delay function $\text{delay}$ and the channel capacity function $\text{size}$ are reused from Definition 3.3 for non-hierarchical network graphs.

In the following, an actor $a_{\text{in}} \in g.A_I$ of a cluster $g$ will be called an input actor if it consumes tokens from channels of its cluster environment. Analogously, an actor $a_{\text{out}} \in g.A_O$ of a cluster $g$ will be called an output actor if it produces tokens on channels of its cluster environment. Actors which are neither input nor output actor are called intermediate actors. More formally, the set of input actors $A_I$ of a cluster $g$ is defined as $g.A_I = \{ a \in g.A | \exists (v_{\text{src}}, v_{\text{snk}}) \in g.E : v_{\text{src}} \in g.I \land v_{\text{snk}} \in a.I \}$, while the set of output actors $A_O$ of a cluster $g$ is defined analogously as $g.A_O = \{ a \in g.A | \exists (v_{\text{src}}, v_{\text{snk}}) \in g.E : v_{\text{snk}} \in g.O \land v_{\text{src}} \in a.O \}$. Note that $A_I$ and $A_O$ may not be disjoint sets.

To distinguish between open and closed systems, the notion of hierarchical network graphs is introduced. These graphs are simply clusters with an empty set of cluster input and output ports.

**Definition 3.5** (Hierarchical Network Graph [FHZT13]). A hierarchical network graph $g_n \in G_n = \{ g | g \in G \land g.P = \emptyset \}$ is a cluster without cluster ports.\(^{26}\)

As can be seen from Definitions 3.4 and 3.5, a non-hierarchical network graph $g_n$, as given in Definition 3.3, is simply a hierarchical network graph without any subclusters, i.e., $g_n.G = \emptyset$. Furthermore, a cluster $g$ will be called non-hierarchical if it does not contain a subcluster, i.e., $g.G = \emptyset$, and hierarchical otherwise. In the following, it will be assumed that a network graph or cluster may contain subclusters if they are not explicitly stated to be non-hierarchical.

### 3.5.3 The Cluster Operation

The transformation of a possibly non-hierarchical original cluster into a hierarchical cluster will be denoted by usage of the $\Gamma : G \times 2^V \rightarrow G \times G$ cluster operator. This operation takes an original cluster and a set of vertices $V_\gamma$ to generate a clustered network graph containing a subcluster which replaces the set of clustered vertices $V_\gamma$. An example of such a cluster operation can be seen

\(^{26}\) In contrast to the usage of $g.G$, which denotes the set of subclusters inside the cluster $g$, or the usage of $G$ where it is apparent from the context to refer to some cluster $g$, the usage of $G$ alone and without any context is used to denote the set of all possible clusters. This convention is additionally used accordingly for the set of all possible actors $A$, channels $C$, edges $E$, ports $P$, and vertices $V = A \cup C \cup G$. 

in Figure 3.6 depicting the original network graph $g_{\text{sq}, \text{flat}}$ and Figure 3.12 displaying the clustered network graph $g_{\text{sq}}$ containing the subcluster $g_{\gamma_{\text{appx}}}$ which replaces the clustered actors and channels $V_{\gamma}^{\text{sq}} = \{ a_3, a_4, c_3, c_4 \} \subseteq g_{\text{sq}, \text{flat}} \cdot V$, i.e., $\Gamma(g_{\text{sq}, \text{flat}}, V_{\gamma}^{\text{sq}}) = (g_{\text{sq}}, g_{\gamma_{\text{appx}}})$.

Without loss of generality, because only actor ports, which are also provided by a subcluster, are referenced inside the following definition, all contained subclusters inside the original graph $g_{\text{orig}}$ are treated like actors. Should an actor be indeed a cluster, then the actor ports will refer to the equivalent cluster ports.

**Definition 3.6 (Cluster Operation [FHT07*]).** The cluster operation $\Gamma : G \times 2^V \rightarrow G \times G$ is a partial function which maps $\Gamma(g_{\text{orig}}, V_{\gamma}) = (g, g_{\gamma})$ an original graph $g_{\text{orig}} \in G$ and a set of channels and actors $V_{\gamma} \subseteq g_{\text{orig}} \cdot V$ contained in this original graph into a clustered graph $g \in G$ and a subcluster $g_{\gamma} \in g \cdot G$ contained in the clustered graph. The cluster operation requires that all channels to cluster $c \in C_{\gamma} = V_{\gamma} \cap g_{\text{orig}} \cdot C$ are connected to actors to cluster $a \in A_{\gamma} = V_{\gamma} \cap g_{\text{orig}} \cdot A$, i.e., $\forall c \in C_{\gamma} : \exists i \in A_{\gamma} \cdot I, o \in A_{\gamma} \cdot O : (c, i), (o, c) \in g_{\text{orig}} \cdot E$. The clustered graph and its subcluster are derived from the original graph $g_{\text{orig}}$ and the set $V_{\gamma}$ of channels and actors to cluster in the following way:
3. SysteMoC

- The vertices $V_\gamma$ are replaced in the clustered graph $g$ by the subcluster $g_\gamma \in g.G$ which contains the replaced vertices $V_\gamma$, i.e., $g_\gamma.V = V_\gamma, g.V = (g_{\text{orig}}.V \setminus V_\gamma) \cup \{ g_\gamma \}$.

- The ports $g.P$ of the clustered graph $g$ are taken from the original graph without any modifications, i.e., $g.P = g_{\text{orig}}.P$.

- The input ports $g_\gamma.I$ of the subcluster $g_\gamma$ are derived via an one-to-one correspondence from the edges $e_{\text{in}} \in E_{\text{in}}$ directed from vertices of the clustered graph to vertices of the subcluster, i.e., $\exists f_{\text{in}} : E_{\text{in}} \rightarrow g_\gamma.I : f_{\text{in}}$ is a bijection where $E_{\text{in}} = \{ e \in g_{\text{orig}}.E \mid e.v_{\text{src}} \in g.C \cup g.I, e.v_{\text{sink}} \in g_\gamma.A.I \}$.

- The output ports $g_\gamma.O$ of the subcluster $g_\gamma$ are derived according to the same scheme as the input ports, i.e., $\exists f_{\text{out}} : E_{\text{out}} \rightarrow g_\gamma.O : f_{\text{out}}$ is a bijection where $E_{\text{out}} = \{ e \in g_{\text{orig}}.E \mid e.v_{\text{src}} \in g_\gamma.A.O, e.v_{\text{sink}} \in g.C \cup g.O \}$.

- After defining the vertices and ports of the clustered graph and its subcluster, the edges connecting these vertices have to be defined. These edges are derived from the edges $e \in g_{\text{orig}}.E$ of the original graph. Two cases have to be distinguished: (1) Edges $e_{\text{in}} \in E_{\text{in}}$ or $e_{\text{out}} \in E_{\text{out}}$ which cross the boundary between the clustered graph and its subcluster and have to be transformed into two edges, one in the clustered graph to or from a subcluster port and one edge in the subcluster from or to a subcluster port, respectively. (2) The remaining edges $e \in g_{\text{orig}}.E \setminus (E_{\text{in}} \cup E_{\text{out}})$ which can be taken verbatim into their corresponding clustered graph and its subcluster. Thus, more formally the edges of the clustered graph are defined as follows: $g.E = \{ e \in g_{\text{orig}}.E \mid e.v_{\text{src}} \in g.P \cup g.C \cup g.A.P \} \cup \{ (e_{\text{in}}, v_{\text{src}}, f_{\text{in}}(e_{\text{in}})) \mid e_{\text{in}} \in E_{\text{in}} \} \cup \{ (f_{\text{in}}(e_{\text{in}}), e_{\text{out}}, v_{\text{sink}}) \mid e_{\text{out}} \in E_{\text{out}} \} \cup \{ (f_{\text{in}}(e_{\text{in}}), e_{\text{in}}, v_{\text{sink}}) \mid e_{\text{in}} \in E_{\text{in}} \} \cup \{ (e_{\text{out}}, v_{\text{src}}, f_{\text{out}}(e_{\text{out}})) \mid e_{\text{out}} \in E_{\text{out}} \}$.

For a better understanding of the preceding definition, the clustering operation $\Gamma(g_{\text{sq}}, \{ a_3, a_4, c_3, c_4 \}) = (g_{\text{sq}}, g_{\text{app}})$ depicted in Figure 3.6 on page 49 and Figure 3.12 is considered again. For this example, the edges crossing the boundary between the clustered graph and its subcluster are $e_{\text{in}} = (e_2, a_3, i_1)$ and $e_{\text{out}} = (a_4, a_2, c_5)$ which are mapped via the bijections $f_{\text{in}}$ and $f_{\text{out}}$ to the subcluster ports $g_{\text{app}}.P = \{ i_1, o_1 \}$ as follows: $f_{\text{in}}(e_{\text{in}}) = g_{\text{app}}.i_1$ and $f_{\text{out}}(e_{\text{out}}) = g_{\text{app}}.o_1$. Furthermore, these two edges are translated into two edges of the clustered graph $(e_{\text{in}} \rightarrow (e_2, g_{\text{app}}.i_1) \in g_{\text{sq}}.E$ and $e_{\text{out}} \rightarrow (g_{\text{app}}.o_1, c_5) \in g_{\text{app}}.P$).

\[27\text{Here the '}'-operator is again used to refer to the source and sink of an edge e, respectively, with the denotation $e.v_{\text{src}}$ and $e.v_{\text{sink}}$, as given in Definition 3.3 and Definition 3.4.}\]
3.6 From SystemC to SysteMoC

While modeling languages with explicit assertions about the used MoC are theoretically very appealing, it is a fact that most models are written in some kind of C or C++-based legacy languages. One of these languages is SystemC [GLMS02, Bai05] which is very popular and used at the ESL as real-world system description language for industrial projects. However, as SystemC is a C++-based class library, it inherits all the problems of C++, e.g., the possibility of unstructured communication over shared memory and Turing completeness. The question answered in this section is whether such legacy code might be transferred into MoCs such as SysteMoC. It is clear that due to Turing completeness, such a transformation can only provide a sufficient criterion to detect a MoC for a given SystemC code. The first step is called communication extraction. The second step to bridge the gap between legacy code and the MoC world will be presented in Section 3.7. Communication extraction, as published also in [FZK+11*], is an algorithm to derive an actor FSM from a SystemC module written in a constrained subset of SystemC.

Fortunately, the synthesizable subset of SystemC [MSS+09] is a good basis for such a constrained subset of SystemC analyzable by the presented methodology. Furthermore, SystemC strongly suggests and the synthesizable subset of SystemC requires the representation of a design in an actor-oriented way [ET06, JBP06]. Designs represented in an actor-oriented way consist of actors which can only communicate via channels connecting the actors via ports, hence alleviating the unstructured communication problem inherited from C++. One key insight is that the number of possible states for the control flow exhibited by a module written in the synthesizable subset of SystemC must be finite. This requirement stems from the observation that behavioral synthesis tools like Forte Cynthesizer [For13], Catapult C from Calypto Design Systems [Cal13], and Vivado from Xilinx [XIL14] represent the control flow of the original behavioral SystemC module as an FSM in the synthesized RTL SystemC module.
However, the synthesizable subset of SystemC is not constrained to use channels exhibiting FIFO semantics. Therefore, synthesizable SystemC applications do not purely consist of actors communicating in a data flow manner in general. Hence, the further constraint that communication must be conducted via FIFO channels is required in order to extract a SysteMoC model from a SystemC application. In practice, SystemC \texttt{sc\_fifo} primitives are used to implement channels of the desired FIFO semantics and limited channel capacity. An example of a SystemC actor satisfying this constraint is depicted in Example 3.4.

Example 3.4 SystemC module implementing the \texttt{a2SqrLoop} actor from Figure 3.6

```cpp
1 // Definition of the SqrLoop actor class
2 class SqrLoop: public sc_core::sc_module { 
3 public:
4   // Declaration of fifo input and output ports
5   sc_core::sc_fifo_in<double> i1, i2;
6   sc_core::sc_fifo_out<double> o1, o2;
7 private:
8   // SystemC thread process
9   void loop() {
10  BB1: double tmp, res;
11  BB2: while (true) {
12  BB3:  tmp = i1.read(); // get token from Src
13  BB4:  do { /* Do one approximation step */
14  BB5:      res = i2.read();
15  BB6:     } while (/* Good enough? */
16  BB7:      std::fabs(tmp-res*res)>=0.01);
17  BB8:     o2.write(res); // write token to Sink
18  BB9:   }
19  BB10: return;
20 }
21 public:
22   SC_HAS_PROCESS(SqrLoop);
23   SqrLoop(sc_core::sc_module_name name)  
24     : sc_core::sc_module(name)  
25   { SC_THREAD(loop); }  
26 27 };
```

Communication extraction [FZK+11*] uses the Control Flow Graph (CFG) of a SystemC module which is extracted via the Abstract Syntax Tree (AST)
generated by the C Language Family Frontend of the LLVM Compiler Infrastructure (clang) [Lat11] parser. The CFG used is very similar to the ones known from compiler design, i.e., it is a directed graph where a vertex represents a Basic Block (BB) and an edge represents a possible successor BB after the BB at the source of the edge has finished execution. A BB itself represents a maximal piece of sequential code which neither contains any jump targets except at its start nor contains any control flow except the possible multiple outgoing edges at its end. In case of multiple outgoing edges of a BB, the edges must be annotated with mutually exclusive Boolean conditions to ensure a deterministic execution semantics.

The communication extraction analysis requires that the SystemC module contains exactly one SystemC thread, i.e., exactly one \texttt{SC\_THREAD(method)} is defined in the module’s constructor and assumes that only functions in the SystemC module itself issue read or write operations on the ports of the module. All other code, e.g., standard library functions, global helper code, and code reachable via function pointers or virtual methods are assumed to have no access to the ports of the SystemC module which calls these functions. The C++ type of a SystemC module is extracted via the usage of C++ Run Time Type Information (RTTI) at run time after the elaboration phase of SystemC. Furthermore, CFGs from compiler design have call and return vertices which enable them to represent infinite recursion. This would prevent the construction of a finite CFG by inlining the CFGs of the called functions. Therefore, any recursion within a SystemC module is also forbidden. The CFG of a given SystemC module is constructed by starting from the \texttt{SC\_THREAD} marked method and is extended by recursively inlining all CFGs of called methods.

Fortunately, the above limitations correspond well to the requirements of the synthesizable subset of behavioral compilers like Forte Cynthesizer [For13] or Catapult C from Calypto Design Systems [Cal13]. This coincidence is not surprising as behavioral compilers also represent the control flow by an FSM.

To exemplify, the SystemC module depicted in Example 3.4 is considered. The module implements the $a_{2}\text{SqrLoop}$ actor from Figure 3.6. The CFG corresponding to the \texttt{loop} method is depicted in Figure 3.13. The CFG deviates from a conventional CFG as known from compiler design, since BBs are split such that all read statements in a BB precede all write statements in the BB. An example of such a split are the BBs BB4 and BB5 which would be a single BB in a traditional CFG. The requirement to split this block stems from the fact that the actions $f_{\text{action}} \in F_{\text{action}}$ will be assembled from the code in the BBs and token consumption and production requirements of an action are checked by the input/output guard at the start of the action. If a BB would contain a read statement after a write statement, an erroneous dependency from the token consumed by the read statement to the token produced by the
write statement would be generated by the input/output guard at the start of the action containing this BB.

It should be noted that the CFG depicted in Figure 3.13 is already annotated with input/output guards for each BB, e.g., the input guard \(#i_1 \geq 1\) on the edge between BB2 and BB3 denotes that at least one token must be available on the FIFO channel connected to the input port \(i_1\) to execute BB3. The CFG is converted into the corresponding actor FSM by converting each BB into a state, e.g., BB1–BB7 is converted to the states \(q_1–q_7\), and adding an initial state \(q_0\). A transition from the initial state \(q_0\) to the state corresponding to the initial BB, e.g., state \(q_1\) derived from BB1, is added. The action for each transition is the BB from which the destination state of the transition is derived. The derived initial actor FSM can be seen in Figure 3.14.

To reduce the number of states and assemble basic blocks into longer actions, a number of post-processing rules may be applied to the initial FSM. These rules will be applied iteratively until no more rules match. In order to express these post-processing rules more succinctly, two transitions \(t_a\) and \(t_b\) are called **sequentially mergeable** if the destination state of transition \(t_a\) is the source state of transition \(t_b\) and \(t_b\) does not consume any tokens if tokens have been produced by \(t_a\). Furthermore, two transitions \(t_a\) and \(t_b\) are called **parallel mergeable** if they have the same source and destination state as well as the same input/output guard. More explicitly, the following post-processing rules [FZK+11*] are applied to the initial actor FSM:

**Rule 1** Elimination of sequentially mergeable transitions for split/join states.

States with at least one incoming transition and exactly one outgoing transition will be called **join states**. States with at least one outgoing
3.6 From SystemC to SysteMoC

Figure 3.14: Transformation of the CFG from Figure 3.13 into an FSM

Figure 3.15: The actor FSM after applying rule 1 to eliminate the states $q_1$, $q_3$, and $q_5$ from Figure 3.14

transition and exactly one incoming transition will be called split states. These states can be eliminated if each incoming transition is sequentially mergeable with each outgoing transition. The state is eliminated by creating a transition $t$ for each combination of incoming transition $t_a$ and outgoing transition $t_b$. The action of the created transition $t$ is the concatenation of the actions of $t_a$ and $t_b$ while the guard is the conjunction of the guards of $t_a$ and $t_b$. The source state of a transition $t$ is the source state of $t_a$ and the destination state of $t$ is the destination state of $t_b$. Examples of such states are $q_1$, $q_3$, $q_5$, and $q_6$ from
Figure 3.14. The resulting FSM after applying this rule to the states $q_1$, $q_3$, and $q_5$ can be seen in Figure 3.15.

Rule 2 Merging of parallel mergeable transitions. The resulting transition $t$ from the parallel transitions $t_a$ and $t_b$ has the same input/output guard as the two original transitions had. The guard function for the resulting merged transition $t$ consists of the (possibly simplified) disjunction of the original guard functions. The resulting action of the merged transition $t$ will consist of an appropriate if construct containing the actions of the original transitions.$^{28}$

Rule 3 Elimination of self-loops. Self loops which contain no input/output guards, i.e., the action on the transition of the self-loop neither consumes nor produces tokens, can be eliminated by concatenating the contained action with an appropriate looping construct for the actions of transitions entering the state with the self-loop under elimination.

Finally, state machine minimization is applied on the resulting transformed actor FSM and subsequently loop unrolling for loops with known boundaries. Note that these are only loops with loop bodies containing read or write operations, due to the elimination of all other loops by the post-processing rules. The resulting FSM after applying the above steps is depicted in Figure 3.16. It can be seen that the resulting FSM strongly resembles the actor FSM from Figure 3.5. The only difference is the presence of the transition from $q_0$ to $q_2$ which is responsible for initializing the variables tmp and res.

3.7 Automatic Model Extraction

In order to be able to apply MoC specific analysis methods such as static schedule generation or deadlock detection, it is important to recognize actors belonging to well-known data flow models of computation such as HSDF, SDF and CSDF. It will be shown that this can be accomplished by inspection and analysis of the actor FSM of a SysteMoC actor. The methodology for actor classification has been published in [FHZT13*, ZFHT08*]. Note that automatic model extraction by means of classification has mainly been researched by Zebelein. However, this step is required for the design flow proposed in this thesis and, hence, will be briefly summarized below.

$^{28}$Note that in the SqrLoop example, this and the following rule will never apply since there are no parallel mergeable transitions or self-loops without communication. A more complex example requiring the application of these rules can be found in [FZK+11*].
3.7 Automatic Model Extraction

\[ q_0 \rightarrow q_0 \]

\[ t/BB1;BB2 \quad \#i_1 \geq 1 \land \#o_1 \geq 1/BB3;BB4 \]

\[ #i_2 \geq 1 \land \#o_2 \geq 1 \land |\text{tmp} - \text{res}^2| \geq 0.01/BB5;BB4 \]

\[ #i_2 \geq 1 \land \#o_2 \geq 1 \land |\text{tmp} - \text{res}^2| < 0.01/BB5;BB6;BB2 \]

**Figure 3.16:** The actor FSM after applying rule 1 to eliminate the states \( q_6 \) from Figure 3.15

The most basic representation of static actors encoded as actor FSMs can be seen in Figure 3.17. To exemplify, the SDF actor depicted in Figure 3.17a corresponds to the \( \text{Dup} \) actor \( a_4 \) as displayed in Figure 3.6. The depicted actor FSM contains only one transition, which consumes one token from the actor input port \( i_1 \) and duplicates this token on the output ports \( o_1 \) and \( o_2 \). Clearly, the actor exhibits a static communication behavior corresponding to the SDF MoC.

**Figure 3.17:** Basic representations of an SDF and a CSDF actors via FSMs

However, in order for the classification algorithm to ignore the guard functions used by an actor FSM, certain assumptions have to be made. It will be assumed that each actor state has at least one enabled outgoing transition if sufficient tokens are present on the actor input ports. This is required in order to be able to activate the equivalent SDF or CSDF actor an infinite number of times. It will also be assumed that an actor will consume or produce tokens every now and then. These assumptions are not erroneous as, otherwise, there exists an infinite loop in the execution of the actor where a cycle of the FSM is traversed and each transition in this cycle neither consumes nor produces any tokens. This can clearly be identified as a bug in the implementation of the actor, similar
to an action of a transition that never terminates. For the exact mathematical
definition of the above given requirements, see [ZFHT08*].

The idea of the classification algorithm is to check if the communication be-
havior of a given actor can be reduced to a basic representation, which can be
easily classified into the SDF or CSDF MoC. Note that the basic representations
for both SDF and CSDF models are FSMs with a single cycle, e.g., as depicted
for the CSDF actor in Figure 3.17b.

Not all actor FSMs exhibit such a regular pattern (cf. Figure 3.18). However,
some of them, e.g., Figures 3.18c and 3.18d, still represent static actors. It can
be distinguished if the analysis of the actor functionality state (cf. Definition 3.1)
is required to decide whether an actor is a static actor, e.g., Figure 3.18c, or not, e.g., Figure 3.18d. In the following, the actor functionality state will not be
considered. Therefore, the presented classification algorithm will fail to classify
Figure 3.18c as a static actor. In that sense, the algorithm only provides a sufficient
criterion for the problem of static actor detection. A sufficient and necessary
criterion cannot be given as the problem is undecidable in the general
case.

The algorithm starts by deriving a set of classification candidates solely on the
basis of the specified actor FSM. Each candidate is later checked for consistency
with the entire FSM state space via Algorithm 1. If one of the classification candidates is accepted by Algorithm 1, then the actor is recognized as a CSDF
actor where the CSDF phases are given by the accepted classification candidate.

Definition 3.7 (Classification Candidate [ZFHT08*]). A possible CSDF behavior of a given actor is captured by a classification candidate \( \mu = \langle \mu_0, \mu_1, \ldots, \mu_{\tau-1} \rangle \)
where each \( \mu = (\text{cons, prod}) \) represents a phase of the CSDF behavior and \( \tau \) is
the number of phases.

To exemplify, Figure 3.17b is considered. In this case, two phases are present,
i.e., \( \tau = 2 \), where the first phase \( (\mu_0) \) consumes one token from port \( i_1 \) and
produces one token on port \( o_1 \) while the second phase \( (\mu_1) \) consumes one token
from port \( i_2 \) and produces one token on port \( o_2 \).

It can be observed that all paths starting from the initial actor FSM state
\( q_0 \) must comply with the classification candidate. Furthermore, as CSDF ex-
hibits cyclic behavior, the paths must also contain a cycle. The classification
algorithm will search for such a path \( p = \langle t_1, t_2, \ldots, t_n \rangle \) of transitions \( t_i \) in the
actor FSM starting from the initial state \( q_0 \). The path can be decomposed
into an acyclic prefix path \( p_a \) and a cyclic path \( p_c \) such that \( p = p_a p_c \), i.e.,
\( p_a = \langle t_0, t_1, \ldots, t_{l-1} \rangle \) being the prefix and \( p_c = \langle t_l, t_{l+1}, \ldots, t_{n-1} \rangle \) being the cyclic
part, that is \( t_l.q_{src} = t_{n-1}.q_{dst} \). After such a path \( p \) has been found, a set
of classification candidates can be derived from the set of all non-empty pre-
fixes \( \{ p' \sqsubseteq p \mid \#p' \in \{ 1, 2, \ldots, n \} \} \) of the path \( p \). A classification candidate
\( \mu = \langle \mu_0, \mu_1, \ldots, \mu_{\tau-1} \rangle \) is derived from a non-empty prefix \( p' \) by (1) unifying
3.7 Automatic Model Extraction

(a) Depicted above is the actor FSM of the SqrLoop actor $a_2$ that is obviously not a static actor.

(b) Visualized above is an actor FSM that cannot be converted to a basic representation for static actors.

(c) Shown is an actor FSM that seems to belong to the dynamic domain, but exhibits CSDF behavior due to the manipulation of the Boolean variable $b$. Hence, leading to a cyclic execution of the transitions $t_1$, $t_3$, and $t_2$.

(d) Shown is an actor FSM that can be converted to a basic representation for static actors.

**Figure 3.18:** Various actor FSMs that do not fit the basic representation for static actors as exemplified in Figure 3.17

adjacent transitions $t_j$, $t_{j+1}$ according to the below given transition contraction condition until no more contractions can be performed and (2) driving the CSDF phases $\mu_j$ of the classification candidate from the transitions $t_j$ of the contracted path computed in step (1).

**Definition 3.8** (Transition Contraction Condition [FHZT13*]). Two transitions $t_j$ and $t_{j+1}$ of a prefix path $p'$ can be contracted if $t_j$ only consumes tokens, i.e., $\text{prod}(t_j, O) = 0$, or $t_{j+1}$ only produces tokens, i.e., $\text{cons}(t_{j+1}, I) = 0$. The resulting transition $t'$ has the combined consumption and production rates given as $\text{cons}(t', I) = \text{cons}(t_j, I) + \text{cons}(t_{j+1}, I)$ and $\text{prod}(t', O) = \text{prod}(t_j, O) + \text{prod}(t_{j+1}, O)$.

For clarification, Figures 3.18b and 3.18d is considered and it is assumed that the path $p = (t_1, t_3, t_4)$ has been found by the classification algorithm.

---

29For notational brevity, the construct $\text{cons}(t, I) = (n_{i_1}, n_{i_2}, \ldots, n_{i_{|I|}}) = \mathbf{n}_{\text{cons}}$ is used to denote the vector $\mathbf{n}_{\text{cons}}$ of numbers of tokens consumed by taking the transition. The equivalent notation $\text{prod}(t, O)$ is also used for produced tokens.
Hence, the set of all non-empty prefixes is \{ \langle t_1 \rangle, \langle t_1, t_3 \rangle, \langle t_1, t_3, t_4 \rangle \}. In both depicted FSMs, the transition \( t_2 \) consumes and produces exactly as many tokens as the transition sequence \( \langle t_3, t_4 \rangle \). However, the transition sequence \( \langle t_3, t_4 \rangle \) in Figure 3.18d can be contracted as \( t_3 \) only consumes tokens while transition sequence \( \langle t_3, t_4 \rangle \) in Figure 3.18b cannot be contracted. Hence, for Figure 3.18d, the classification candidate \( \mu = \langle \mu_0, \mu_1 \rangle \) derived from the prefix \( p' = \langle t_1, t_3, t_4 \rangle \) is as follows:

\[
\begin{align*}
\mu_{0,\text{cons}} &= \text{cons}(t_1, I) = (1, 0) \\
\mu_{0,\text{prod}} &= \text{prod}(t_1, O) = (1, 0) \\
\mu_{1,\text{cons}} &= \text{cons}(t_3, I) + \text{cons}(t_4, I) = (0, 1) \\
\mu_{1,\text{prod}} &= \text{prod}(t_3, O) + \text{prod}(t_4, O) = (0, 1)
\end{align*}
\]

On the other hand, for Figure 3.18b, the classification candidate \( \mu = \langle \mu_0, \mu_1, \mu_2 \rangle \) derived from the prefix \( p' = \langle t_1, t_3, t_4 \rangle \) does not exhibit any contractions and is depicted below as:

\[
\begin{align*}
\mu_{0,\text{cons}} &= (1, 0) & \mu_{0,\text{prod}} &= (1, 0) \\
\mu_{1,\text{cons}} &= (0, 0) & \mu_{1,\text{prod}} &= (0, 1) \\
\mu_{2,\text{cons}} &= (0, 1) & \mu_{2,\text{prod}} &= (0, 0)
\end{align*}
\]

The underlying reasons for the transition contraction condition from Definition 3.8 is discussed in the following. To illustrate, the data flow graph depicted in Figure 3.19a which uses two actors \( a_2 \) and \( a_3 \) containing the FSMs from Figures 3.18b and 3.18d is considered. The resulting dependencies between the transitions in a transition trace of the actors \( a_2 \) and \( a_3 \) are shown in Figure 3.19b.

As can be seen in Figure 3.19c if the transition sequence \( \langle t_3, t_4 \rangle \) of actor \( a_3 \) is contracted into a single transition \( t_c \), then the resulting dependencies of \( t_c \) are exactly the same as for transition \( t_2 \). This is the reason why the FSM from Figure 3.18d can be classified into a CSDF actor with an actor FSM as depicted in Figure 3.17b. Furthermore, the contraction is a valid transformation as it does not change the data dependencies in the data flow graph. This can be seen by comparing the transition dependencies depicted in Figures 3.19b and 3.19c. Compacting the transition sequence \( \langle t_3, t_4 \rangle \) of actor \( a_3 \) generates the transition \( t_c \), which has a data dependency from the consumption of a token on \( i_2 \) to the production of a token on \( o_2 \). However, the previous transition sequence \( \langle t_3, t_4 \rangle \) also induces this data dependency as \( t_4 \) can only be taken after \( t_3 \).

In contrast to this, the contraction of the transition sequence \( \langle t_3, t_4 \rangle \) of actor \( a_2 \) into a transition \( t_d \) does introduce a new erroneous data dependency from the consumption of a token on \( i_2 \) to the production of a token on \( o_2 \). The data dependency is erroneous as the original transition sequence \( \langle t_3, t_4 \rangle \) has no such dependency as it first produces the token on \( o_2 \) before trying to consume a token on \( i_2 \). Indeed, an erroneous contraction might introduce a deadlock.
shown is a DFG containing FSMs from Figure 3.18b (in actor $a_2$) and Figure 3.18d (in actor $a_3$). Due to data dependencies, actor $a_2$ will never execute transition $t_2$. On the other hand, actor $a_3$ is free to choose either the transition sequence $⟨t_3, t_4⟩$ or the transition $t_2$ in its execution trace.

(b) Shown are the dependencies in the transition trace of actor $a_2$ and $a_3$. Dashed lines represent data dependencies. Solid lines represent dependencies induced by the sequential nature of the FSM.

(c) Shown are the dependencies for the contraction of the transition sequence $⟨t_3, t_4⟩$ of actor $a_3$ into the transition $t_c$. Note that transition $t_2$ and the contracted transition $t_c$ of actor $a_3$ have the same data dependencies.

(d) Shown are the dependencies for the contraction of the transition sequence $⟨t_3, t_4⟩$ of actor $a_2$ into the transition $t_d$. Thus, introducing two erroneous dependency cycles $a_2.t_d → a_3.t_3 → a_3.t_4 → a_2.t_d$ and $a_2.t_d → a_3.t_3 → a_3.t_4 → a_2.t_d$.

Figure 3.19: Depicted above is a DFG containing two actors with FSMs shown in Figures 3.18b and 3.18d as well as the dependencies in their transition traces after various transition sequences have been contracted.

75
Algorithm 1 Validation of a classification candidate $\mu$ for the actor FSM $\mathcal{R}$

1: function VALIDATECLASSCAND($\mu, \mathcal{R}$) 
2: $\tau = \#\mu$ \hfill \triangleright \text{Number of CSDF phases } \tau 
3: $\langle \mu_0, \mu_1, \ldots, \mu_{\tau-1} \rangle = \mu$ \hfill \triangleright \text{CSDF phases } \mu_0, \mu_1, \ldots, \mu_{\tau-1} 
4: $\omega_0 = (0, \mu_0.\text{cons}, \mu_0.\text{prod})$ \hfill \triangleright \text{Initial annotation tuple} 
5: $\text{queue} \leftarrow \langle \rangle$ \hfill \triangleright \text{Set up the empty queue for breadth-first search} 
6: $\text{ann} \leftarrow \emptyset$ \hfill \triangleright \text{Set up the empty set of annotations} 
7: $\text{queue} \leftarrow \text{queue}^\triangleright (\mathcal{R}.q_0)$ \hfill \triangleright \text{Enqueue } \mathcal{R}.q_0 
8: $\text{ann} \leftarrow \text{ann}\cup\{ (\mathcal{R}.q_0, \omega_0) \}$ \hfill \triangleright \text{Annotate the initial tuple} 
9: while $\#\text{queue} > 0$ do \hfill \triangleright \text{While the queue is not empty} 
10: $q_{\text{src}} = \text{queue}(0)$ \hfill \triangleright \text{Get head of queue} 
11: $\omega_{\text{src}} = \text{ann}(q_{\text{src}})$ \hfill \triangleright \text{Get annotation tuple for state } q_{\text{src}} 
12: $\text{queue} \leftarrow \text{tail}(\text{queue})$ \hfill \triangleright \text{Dequeue head from queue} 
13: for all $t \in \mathcal{R}.T$ where $t.q_{\text{src}} = q_{\text{src}}$ do 
14: if $\omega_{\text{src}}.\text{cons} \ngeq \text{cons}(t, I) \lor \omega_{\text{src}}.\text{prod} \ngeq \text{prod}(t, O)$ then 
15: return $f$ \hfill \triangleright \text{Reject } \mu \text{ due to failed transition criterion I} 
16: if $\omega_{\text{src}}.\text{cons} \neq \text{cons}(t, I) \land \text{prod}(t, O) \neq 0$ then 
17: return $f$ \hfill \triangleright \text{Reject } \mu \text{ due to failed transition criterion II} 
18: $\omega_{\text{dst}} \leftarrow \text{derive from } \omega_{\text{src}} \text{ and } t \text{ as given by Equation (3.1)}$
19: if $\exists \omega'_{\text{dst}} : (t.q_{\text{dst}}, \omega'_{\text{dst}}) \in \text{ann}$ then \hfill \triangleright \text{Annotated tuple present?} 
20: if $\omega'_{\text{dst}} \neq \omega_{\text{dst}}$ then \hfill \triangleright \text{Check annotated tuple for consistency} 
21: return $f$ \hfill \triangleright \text{Reject classification candidate } \mu 
22: else \hfill \triangleright \text{No tuple annotate to state } t.q_{\text{dst}} 
23: $\text{ann} \leftarrow \text{ann}\cup\{ (t.q_{\text{dst}}, \omega_{\text{dst}}) \}$ \hfill \triangleright \text{Annotate tuple } \omega_{\text{dst}} 
24: $\text{queue} \leftarrow \text{queue}^\triangleright (t.q_{\text{dst}})$ \hfill \triangleright \text{Enqueue } t.q_{\text{dst}} 
25: return $t$ \hfill \triangleright \text{Accept classification candidate } \mu 

The main idea of the validation algorithm is to check the classification candidate against all possible transition sequences reachable from the initial state of the actor FSM. However, due to the existence of contracted transitions in the classification candidate as well as in the actor FSM, a simple matching of a phase $\mu_j$ to an FSM transition is infeasible. Instead, a CSDF phase $\mu_j$ is matched by a transition sequence. To keep track of the number of tokens already produced and consumed for a CSDF phase, each FSM state $q_n$ will be annotated with a tuple $\omega_n = (j, \text{cons, prod})$ where cons and prod are the vectors of number of tokens which are still to be consumed and produced to complete the consumption and production of the CSDF phase $\mu_j$ and $j$ denotes the CSDF phase which should be matched.

The validation algorithm starts by deriving the tuple $\omega_0$ (cf. Line 4) for the initial state $q_0$ from the first CSDF phase $\mu_0$ of the classification candidate $\mu$. 

76
3.7 Automatic Model Extraction

The algorithm uses the set \( \text{ann} \) as a function \( \omega_n = \text{ann}(q_n) \) from an FSM state to its annotated tuple \( \omega_n \). Initially, the \( \text{ann} \) set is empty (cf. Line 6) denoting that all function values \( \text{ann}(q_n) \) are undefined, and hence no tuples have been annotated. The annotation of tuples starts with the initial tuple \( \omega_0 = \text{ann}(q_0) \) for the initial state \( q_0 \) by adding the corresponding association to the \( \text{ann} \) set in Line 8.

The algorithm proceeds by performing a breadth-first search (cf. Lines 5, 9, 12, and 24) of all states \( q_n \) of the FSM \( \mathcal{R} \) starting from the initial state \( q_0 \) (cf. Line 7). For each visited state \( q_{\text{src}} \) (cf. Line 10) its annotated tuple \( \omega_{\text{src}} \) will be derived from the set \( \text{ann} \) (cf. Line 11) and the corresponding outgoing transitions \( t \) of the state \( q_{\text{src}} \) (cf. Line 13) are checked against the annotated tuple \( \omega_{\text{src}} \) (cf. Lines 14 to 17) via the transition criteria as given below:

**Definition 3.9** (Transition Criterion I [ZFHT08*]). Each outgoing transition \( t \in T_{\text{src}} = \{ t \in T \mid t.q_{\text{src}} = q_{\text{src}} \} \) of a visited FSM state \( q_{\text{src}} \in Q \) must consume and produce less or equal tokens than specified by the annotated tuple \( \omega_{\text{src}} \), i.e., \( \forall t \in T_{\text{src}} : \omega_{\text{src}}.\text{cons} \geq \text{cons}(t, I) \land \omega_{\text{src}}.\text{prod} \geq \text{prod}(t, O) \). Otherwise, the annotated tuple \( \omega_{\text{src}} \) is invalid and the classification candidate \( \mu \) will be rejected.

**Definition 3.10** (Transition Criterion II [ZFHT08*]). Each outgoing transition \( t \in T_{\text{src}} = \{ t \in T \mid t.q_{\text{src}} = q_{\text{src}} \} \) of a visited FSM state \( q_{\text{src}} \in Q \) must not produce tokens if there are still tokens to be consumed in that phase after the transition \( t \) has been taken, i.e., \( \forall t \in T_{\text{src}} : \omega_{\text{src}}.\text{cons} = \text{cons}(t, I) \lor \text{prod}(t, O) = 0 \). Otherwise, the annotated tuple \( \omega_{\text{src}} \) is invalid and the classification candidate \( \mu \) will be rejected.

*Transition criterion I* ensures that a matched transition sequence consumes and produces exactly the number of tokens as specified by the CSDF phase \( \mu \) of the classification candidate. *Transition criterion II* ensures that a transition sequence induces the same data dependencies as the CSDF phase \( \mu \) of the classification candidate. If the transition does not conform to the above transition criteria, then the classification candidate is invalid and will be rejected (cf. Lines 15 and 17). Otherwise, that is conforming to both criteria, the matched transition sequence can be condensed via Definition 3.8 to the matched CSDF phase.

After the transition has been checked, the tuple \( \omega_{\text{dst}} \) for annotation at the transition destination state \( t.q_{\text{dst}} \) is computed in Line 18 according to the equation given below:

\[
\begin{align*}
\text{cons}_{\text{left}} &= \omega_{\text{src}}.\text{cons} - \text{cons}(t, I) \\
\text{prod}_{\text{left}} &= \omega_{\text{src}}.\text{prod} - \text{prod}(t, O) \\
\tau &= (\omega_{\text{src}}.j + 1) \mod \tau \\
\omega_{\text{dst}} &= \begin{cases} 
(\omega_{\text{src}}.j, \text{cons}_{\text{left}}, \text{prod}_{\text{left}}) & \text{if } \text{cons}_{\text{left}} \neq 0 \land \text{prod}_{\text{left}} \neq 0 \\
(j', \mu_{j'}.\text{cons}, \mu_{j'}.\text{prod}) & \text{otherwise}
\end{cases}
\end{align*}
\] (3.1)

77
3. SysteMoC

In the above equation, $\text{cons}_{\text{left}}$ and $\text{prod}_{\text{left}}$ denote the consumption and production vectors remaining to match the CSDF phase $\mu_j$. If these remaining consumption and production vectors are both the zero vector, then the matching of the CSDF phase $\mu_j$ to a transition sequence has been completed. Hence, the tuple $\omega_{\text{dst}}$ will be computed to match the next CSDF phase $\mu_{(j+1) \mod r}$. Otherwise, the tuple $\omega_{\text{dst}}$ will use the updated remaining consumption and production vectors as given by $\text{cons}_{\text{left}}$ and $\text{prod}_{\text{left}}$.

Finally, if no tuple is annotated to the destination state $t.q_{\text{dst}}$, then the computed tuple $\omega_{\text{dst}}$ is annotated to the destination state in Line 23 and the destination state is appended to the queue for the breadth-first search in Line 24. Otherwise, an annotated tuple is already present for the destination state (cf. Line 19). If this annotated tuple $\omega'_{\text{dst}}$ and the computed tuple $\omega_{\text{dst}}$ are inconsistent, then the classification candidate will be rejected (cf. Lines 19 to 21).

3.8 Related Work

The POLIS [BCG+97] approach is focused on control dominated applications. The internal representation of POLIS is the Codesign Finite State Machine (CFSM). The POLIS approach supports the import of these CFSMs from input languages that are based on an inherent FSM semantics like Esterel [BCE+03] or the synthesizable subsets of VHDL and Verilog. A POLIS application is assembled from these CFSMs by connecting them in an asynchronous way, whereas each CFSM is executed in a synchronous manner. Hence, POLIS has been designed to model Globally Asynchronous Locally Synchronous (GALS) systems. The asynchronous communication between different CFSMs is not constrained to use data flow semantics, but is implemented via finite buffers which can be overwritten. Hence, no guarantee of a successful message delivery can be given. The POLIS approach supports the generation of C and VHDL code from the CFSMs and their connection topology, however, the mapping decision for each CFSM to either software or hardware has to be provided manually.

YAPI [dKSvdW+00] is a C++ class library that implements an API for simulation of Dennis Data Flow (DDF) data flow graphs. Hence, YAPI applications are not compositional and cannot be used to model applications from the packet processing domain. Applications of the packet processing domain require non-determinacy, which is missing from the DDF model employed by YAPI, in order to react to their environment. Hence, YAPI is aimed at modeling applications of the signal processing domain. The YAPI approach does not provide any synthesis or code transformation back-ends for applications modeled with the YAPI API.

---

30 The notation $n = l \mod m$ for values $l \in \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, $m \in \mathbb{N} = \{1, 2, \ldots\}$ is used to denote the common residue $n \in \{0, 1, 2, \ldots, m - 1\}$ which is a non-negative integer smaller than $m$ such that $l \equiv n \pmod{m}$. 
Ptolemy II [EJL+03, Pto14], the successor of Ptolemy Classics [BHLM94], is a modeling framework that focuses on the integration of various MoCs. The Ptolemy approach achieves this integration by allowing the refinement of one actor contained in a certain MoC by a graph of actors under a different MoC. Hence, Ptolemy can be thought of a continuation and refinement of the *charts (pronounced "star charts") [GLL99, Pto14] approach presented in Section 2.4. Thus, in Ptolemy each actor is associated with a MoC by a director which explicitly gives the MoC under which the actor is executed. Furthermore, the behavior of an actor might change depending on the director under which the actor is executed. As far as code generation is concerned, Ptolemy II up to versions 8.0 supported the generation of C and Java code for non-hierarchical SDF models by the Copernicus code generator. However, this generator has been removed in recent Ptolemy II releases and code generation is currently not a focus of the Ptolemy research effort.

While Ptolemy itself is a pure modeling environment without support for modeling of possible architectures and their influence on the execution of the functionality, an extension of Ptolemy called PeaCE [HKL+07] (Ptolemy extension as Codesign Environment) provides this support. The PeaCE codesign environment is based on the Synchronous Piggybacked Data Flow (SPDF) MoC [PCH99], an extension of SDF enabling the efficient distribution of control information by piggybacking this information onto data tokens. PeaCE itself has no dedicated input language, but specifies its models by files containing the topology of the SPDF data flow graph and a library based approach to specify the actors.

Metropolis [BWH+03] and Metropolis II [DDG+13] are representatives of the modeling environment using platform-based design. Platform-based design is characterized by a two-level approach consisting of an architecture providing services to a functionality. The platform is defined by the set of services provided by the architecture. The functionality, which is modeled as concurrent processes, is not mapped directly to the architecture, but is adapted to the set of services provided by the platform. Hence, a different architecture implementing the same platform will also be able to run the functionality. The modeling in Metropolis is separated into processes representing the functionality, media representing the services provided by the platform, and so-called quantity managers to arbitrate access to the media by the functionality. Hence, different realizations of the platform via different architectures are modeled by metropolis by using different quantity managers. Quantity managers not only arbitrate the access of the media, but also annotate each access of the media by the functionality with the corresponding resource costs like execution time or power consumption.

HetSC (Heterogeneous SystemC) [HV08, HVG+08] is a C++ class library extending SystemC to support the specification of domains realizing different MoCs. HetSC supports the specification of data flow MoCs like KPN [Kah74],
KPN with bounded FIFO capacities, and SDF [LM87a] as well as none data flow MoCs like Synchronous Reactive (SR) [BCE'03, BB91] and Communicating Sequential Processes (CSP) [Hoa78, Hoa85]. However, HetSC does not provide any support to extract a mathematical model of a functionality given by an executable specification modeled in HetSC. To specify a domain realizing a certain MoC, the HetSC methodology defines a set of coding standards for each supported MoC. A domain in HetSC consists of a set of actors and their connecting channels that are constrained to have semantics corresponding to the MoC of the domain. To implement a certain MoC, the HetSC approach maps the required kind of event [LSV98] to implement the MoC into the strict-time Discrete Event (DE) [Yat93, Lee99] domain supported by the SystemC simulation kernel. Hence, the SystemC simulation kernel does not need to be extended with additional solvers for these MoCs in order for the HetSC library to provide support for these MoCs. Thus, the reference implementation of SystemC provided by OSCI can be used unmodified by the HetSC library.

In contrast to this, SystemC-H [PS04] provides support for the SDF, CSP, and FSM MoCs by extending the SystemC simulation kernel itself with dedicated solvers for each of the three MoCs. This enables SystemC-H to provide higher performance simulation engines for its supported MoCs as compared to HetSC. However, modifying the SystemC library itself is detrimental to compatibility as various EDA vendors require the usage of their own SystemC library in order to guarantee a working tool flow.

HetMoC [ZSJ10] is concerned with the integration of the Continuous Time (CT) MoC into SystemC. HetMoC also supports integration of SR, SDF, and the DE models. However, the main focus of HetMoC is an improved integration of the CT domain into SystemC, thus, providing better solvers for analog systems modeled in SystemC as compared to the SystemC-AMS [BEG13] extension.

**Summary**

In this chapter, the first key contribution of this work, the SysteMoC modeling language for the ESL was presented. This language has the ability to model executable specifications which can act as an input model for DSE as well as serve as an evaluator for DSE tools to evaluate timing and power characteristics of the executable specification for the different design decisions that are evaluated by the DSE. The SysteMoC language has strong formal underpinnings in data flow modeling with the distinction that the expressiveness of the data flow model used by an actor is not chosen in advance but determined from the implementation of the actor via classification. In order for the classification to determine the data flow model used by a SysteMoC actor, the SysteMoC
language enforces a distinction between communication and computation of an actor by using an FSM to specify the communication behavior of an actor.

The identification of static actors by means of classification enables the second key contribution of this thesis, the clustering methodology which will be presented in the following. Clustering is a refinement of a SysteMoC data flow graph by replacing islands of static actors in the data flow graph by composite actors. Later, in the software implementation generated by the synthesis back-end, the remaining actors which are not replaced by a composite actor as well as all composite actors are scheduled by a dynamic scheduler at run time. Thus, by combining multiple static actors into one composite actor, the clustering methodology reduces the number of actors which have to be scheduled by the dynamic scheduler. This reduction is of benefit as it has been shown that compile-time scheduling of static actors produces more efficient implementations in terms of latency and throughput than run-time scheduling of the same actors by a dynamic scheduler.
In this chapter, the second key contribution of this work, a clustering methodology [FKH+08*, FZK+11*, FHZT13*] that exploits the presence of highly analyzable static actors in a Data Flow Graph (DFG) even in the presence of dynamic actors, will be presented. This contribution enables an automatic and correct clustering of SystemC Models of Computation (SysteMoC) applications to reduce the number of dynamic scheduling decisions and, thus, enables the generation of more efficient code for Multi-Processor System-on-Chip (MPSoC) targets.

In detail, actors are called static if they conform to one of the following three data flow models Homogeneous (Synchronous) Data Flow (HSDF) [CHEP71], Synchronous Data Flow (SDF) [LM87b], or Cyclo-Static Data Flow (CSDF) [BELP96] introduced in Chapter 2. The clustering methodology is neither limited to DFGs consisting only of static actors, nor does it require a special treatment of these static actors by the developer of the SysteMoC application. Indeed, static actors are identified in an executable specification modeled in the SysteMoC language by means of classification as presented in the previous chapter. This classification enables the clustering methodology to refine the DFG by replacing islands of static actors by composite actors. Islands of static actors, also called static clusters, are connected subgraphs of static actors in the whole DFG described by an executable specification modeled in SysteMoC. A composite actor implements a Quasi-Static Schedule (QSS) for the actors contained in its corresponding static cluster. Intuitively, a QSS is a schedule in which a relatively large proportion of scheduling decisions have been made at compile time. A QSS executes sequences of statically scheduled actor firings at run time. These sequences have been determined at compile time by only postponing required scheduling decisions to run time. Advantages of QSSs include lower scheduling overhead and improved predictability compared to dynamic scheduling at run time. Later, in the implementation generated by the synthesis back-end, each computation resource, e.g., a Central Processing Unit (CPU) or a hardware accelerator resource that supports the mapping of multiple actors to itself [ZFHT12*], possesses its own dynamic scheduler that schedules the actors that are mapped to its resource. After clustering and refinement has been
4. Clustering

performed, the entities to be scheduled at run time by a dynamic scheduler of a resource are the actors bound to this resource that have not been replaced by a composite actor as well as the composite actors themselves that have been bound to this resource. Thus, by combining multiple static actors into one composite actor, the clustering methodology reduces the number of actors which have to be scheduled by the dynamic schedulers. This reduction is of benefit as it has been shown [BBHL95, BL93, BML97] that compile-time scheduling of static actors [LM87b] produces more efficient implementations in terms of latency and throughput than scheduling of the same actors by a dynamic scheduler at run time.

In the following, the structure and contributions of this chapter are outlined: In Section 4.1, dynamic scheduling and the causes of scheduling overhead introduced by this scheduling strategy will be discussed. Next, the required conditions for a refinement replacing a static cluster by a composite actor to be safe will be discussed in Section 4.2. In contrast to this, unsafe refinements might introduce deadlocks into the refined DFGs that result from replacing static clusters by composite actors. Subsequently, in Section 4.3, refinements of a DFG that are constrained to only replace static clusters by composite actors exhibiting SDF or CSDF semantics will be considered. A refinement to a composite actor exhibiting SDF semantics is also called monolithic clustering [TBRL09].

In contrast to this, the second key contribution—the clustering methodology introduced in Section 4.4—goes significantly beyond previous approaches [BL93, PBL95] as it is not confined to monolithic clustering, but produces a quasi-static scheduling interface between the generated composite actor and the cluster environment. This enhances the power of previous clustering techniques in terms of (1) deadlock avoidance, (2) increasing the design space for cluster refinements, and (3) providing the integration of a cluster with its cluster environment. A precise definition of this design space as well as a heuristic to find static clusters amenable to refinement by the introduced clustering methodology will be given in Section 4.4.1. Next, an algorithm [FKH*08*, FZK*11*, FHZT13*] to compute a Finite State Machine (FSM) based representation of a QSS for a static cluster will be given in Section 4.4.2. In contrast to the QSSs computed by the algorithm presented by Tripakis et al. [TBRL09, TBG*13], the algorithms employed by the introduced clustering methodology will even generate retiming sequences. Retiming sequences are used to increase the length of the sequences of statically scheduled actor firings executed by a QSS at run time. Intuitively, the larger the length of a sequence of statically scheduled actor firings is, the lower the scheduling overhead will be at run time. Moreover, an improved rule-based representation of QSSs [FZHT11*, FZHT13*] will be introduced in Section 4.4.3. The rule-based representation trades latency and throughput in order to achieve a more compact code size of a QSS as compared to the automata-based representation. Both algorithms introduced in Sections 4.4.2
4.1 Clustering and Scheduling Overhead

and 4.4.3 encode actor firings that should be statically scheduled as partial repetition vectors. Subsequently, in Section 4.4.4, the partial repetition vectors will be statically scheduled by transforming each partial repetition vector into a looped schedule [HB07]. In contrast to a partial repetition vector, a looped schedule has a direct correspondence to the sequential code that is emitted by the synthesis back-end presented in Chapter 5 in order to realize the execution of a sequence of statically scheduled actor firings. To conclude the theoretical aspects of the introduced clustering methodology, a correctness proof [FZHT13*] will be given in Section 4.4.5. In order to extend the design space for refinement of static clusters into composite actors, a First In First Out (FIFO) channel capacity adjustment algorithm is presented in Section 4.4.6. This algorithm adjusts the channel capacities of the FIFO channels used to connect a static cluster to its cluster environment. This enables—in contrast to the approach presented by Tripakis et al. [TBG+13]—the introduced clustering methodology to be applicable to dynamic DFGs with finite channel capacities instead of channel capacities that are conceptionally considered to be unbounded.

In Section 4.5, the benefits obtained by applying the introduced clustering methodology will be shown. Finally, the related work will be discussed in Section 4.6. Note that parts of this chapter are derived from [FHT07*, FKH*08*, FZK+11*, FZHT11*, FZHT13*].

4.1 Clustering and Scheduling Overhead

In this section, scheduling in the context of SYSTEMCoDESIGNER will be briefly introduced. Then, the relevant reasons for scheduling overhead will be outlined, and it will be discussed how clustering is used to reduce this overhead. Thus, the notion of cluster FSM will be introduced to represent the QSSs that are generated by the proposed clustering methodology.

Scheduling is the problem of determining what job to execute on what resource at which point in time in order to meet demands in a cost-effective and timely manner. However, in case of a design flow using the SYSTEMCoDESIGNER [HFK+07*, KSS+09*] methodology, the binding of actors to resources (cf. Figure 4.1) has already been determined via Design Space Exploration (DSE). Thus, scheduling in this context is concerned with determining for each resource a sequence of actor firings of actors bound onto this resource. Furthermore, it is assumed that an actor firing once started on a resource, cannot be preempted by another actor firing on the same resource.

Strategies for determining a sequence of actor firings on a resource can be distinguished into (1) dynamic scheduling strategies that postpone all scheduling decisions to a dynamic scheduler to be executed at run time and (2) static scheduling strategies that compute all scheduling decisions at compile time.
4. Clustering

Maping edge denoting that actor $a_1$ will be implemented by resource $r_1$

**Figure 4.1:** Depicted above is an implementation graph of an implementation of Newton’s iterative algorithm for calculating the square root using only a dedicated CPU to provide all the computation needs of the algorithm. The network graph $g_{sqr, flat}$ consists of static (shaded) and dynamic (white) actors. The depicted implementation graph is a replication of a solution already shown and described in the previous chapter in Figure 3.4c on page 41.

However, static scheduling can only be applied if the DFG consists solely of static actors. Otherwise, the presence of a dynamic actor introduces behavior into a DFG that cannot be predicted at compile time. Hence, not all scheduling decisions can be predetermined at compile time. On the other hand, dynamic scheduling might introduce a significant scheduling overhead. Scheduling overhead results from the requirement of the dynamic scheduler to determine at run time a sequence of actor firings for a resource that conforms to the data dependencies between these actors. In contrast to this, if a static schedule for these actors is available, then this static schedule can simply be executed at run
time. An example of a dynamic scheduler implementing a dynamic scheduling strategy for all actors of a DFG is given in Algorithm 2.31

**Algorithm 2** Dynamic scheduler for a DFG \( g \)

1: function `DYNAMICSCHEDULER(\( g \))`
2: \( \text{progress} = t \) \( \triangleright \) Mark DFG as active
3: while \( \text{progress} \) do \( \triangleright \) While the DFG is active
4: \( \text{progress} \leftarrow f \) \( \triangleright \) Reset the active state
5: for all \( a \in g.A \) do \( \triangleright \) Iterate over all actors of the graph \( g \)
6: if actor \( a \) is enabled then \( \triangleright \) Check if actor \( a \) can be fired
7: \( \text{progress} \leftarrow t \) \( \triangleright \) Mark DFG as active
8: fire actor \( a \) \( \triangleright \) Fire the actor \( a \)

The dynamic scheduling overhead is induced by (1) checks for the ability of an actor to be fired (cf. Line 6) and (2) deficiencies in code generation for the actions of the actors that are executed (cf. Line 8) by the dynamic scheduler. These deficiencies result from the inability of the compiler to determine that tokens produced by a source actor could immediately be consumed by another sink actor. Hence, token values might not need to be stored in a FIFO channel, but could be kept in a register. Indeed, these token values might not even need to be computed by the source actor if they would be discarded by the sink actor, e.g., a down-sample actor known from the signal processing domain.

To exemplify, consider the implementation of Newton’s iterative algorithm for calculating the square root using only a dedicated CPU as depicted in Figure 4.1. To coordinate the executions of the actors \( a_1 - a_5 \) on the CPU \( r_1 \), a scheduling strategy has to be selected for the CPU. Assuming that the dynamic scheduler depicted in Algorithm 2 is used to schedule the actors of the network graph \( g_{\text{sqr,flat}} \), then a possible schedule of these actors is visualized by the Gantt chart shown in Figure 4.2. This Gantt chart only shows a possible schedule up to the point in time at which the first token arrives at the Sink actor \( a_5 \). Shaded boxes in the Gantt chart correspond to execution time (cf. Line 8) spent to compute Newton’s iterative algorithm for calculating the square root, while the white boxes correspond to execution time (cf. Line 6) spent by the dynamic scheduler in order to coordinate the firings of the actors \( a_1 - a_5 \) on the CPU \( r_1 \).

Each box in the Gantt chart is annotated with the action or guard that spends the execution time represented by the box. For the SqrLoop actor \( a_2 \), the execution of actions and functionality guards has been made explicit by annotating the corresponding function from the actor functionality \( a_2.F = \{ f_{\text{copyStore}}, f_{\text{copyInput}}, f_{\text{copyApprox}}, f_{\text{check}} \} \) (cf. Figure 3.5 on page 45). For the other actors of the network graph \( g_{\text{sqr,flat}} \), the notation \( f_{(a_n)} \) will be used to

---

31 In the following, \( t \) and \( f \) are used to denote Boolean truth values.
denote that the actor \( a_n \) will be executed once by selecting an appropriate action from the actor functionality of the actor \( a_n \). Correspondingly, the notation \( a_n.k_{\#c_i \geq 1}^{\#c_o \geq 1} \) will be used to denote that the dynamic scheduler checks the availability of tokens and free places at the channels \( c_i \) and \( c_o \) connected to the input and output ports of the actor \( a_n \) in order to determine (cf. Line 6) if the actor can be fired. More precisely, a white box annotated with \( k \) denotes the execution time spent for the evaluation of the input/output part of a transition guard \( k \) (cf. Definition 3.2 on page 46).

Each schedule round depicted in Figure 4.2 corresponds to one iteration of the loop body shown in Lines 4 to 8. Failed checks for the ability of an actor to be fired, i.e., consecutive white boxes in the Gantt chart, are especially detrimental to scheduling efficiency.

![Gantt chart](image)

**Figure 4.2:** Gantt chart of a possible schedule of the network graph \( \gamma_{\text{flat}} \) by the dynamic scheduler depicted in Algorithm 2 on the CPU \( r_1 \).

More precisely, the scheduling overhead contained in a possible schedule can be defined as follows:

\[
\frac{t_{SA}}{t_{SA} + t_{TA}}
\]

In this equation, \( t_{SA} \) denotes the overall scheduling time and \( t_{TA} \) denotes the overall computation time. The overall scheduling time is the sum of all execution

88
4.1 Clustering and Scheduling Overhead

times spent by the dynamic scheduler to select transitions of actors for execution. In contrast to this, the overall computation time is the sum of all execution times spent by actors to execute their actions $f_{\text{action}} \in F_{\text{action}}$ and functionality guards $f_{\text{guard}} \in F_{\text{guard}}$, as well as a fixed overhead $t_{TE}$ spent by the dynamic scheduler per execution of one transition. The exact definitions of these times and methods to determine their values can be found in [KSH+11, KSH+12].

Note that the execution times of the functionality guards have been charged to $t_{TA}$ because a functionality guard of an actor only needs to be evaluated at most once per execution of an action of the actor. Considering the definition (cf. Definition 3.1 on page 44) $f_{\text{guard}} : Q_{\text{func}} \times S_{\text{impl}} \rightarrow \{ t, f \}$ of a functionality guard, it can be determined that a functionality guard only depends on the values of the tokens present at the actor input ports and on the functionality state. If no transition of an actor is taken, then neither can values of tokens present at the actor input port change, nor can the functionality state of the actor be modified. Hence, the valuation of the functionality guard will remain unchanged and the result can be cached until a transition of the actor is taken.

To exemplify, consider the Gantt chart depicted in Figure 4.2. If the fixed overhead $t_{TE}$ is neglected, then the overall scheduling time $t_{SA}$ can be computed as the sum of execution times spent by all white boxes and the overall computation time $t_{TA}$ as the sum of execution times spent by all shaded boxes. Moreover, the overall execution time spent by the CPU $r_1$ to execute the DFG till the first token is processed by actor $a_5$ is $t_{EXEC} = t_{SA} + t_{TA} = 61 + 98 = 159$ abstract time units. Hence, for the schedule depicted in Figure 4.2, the following scheduling overhead can be derived:

$$\frac{t_{SA}^{\text{dyn}}}{t_{EXEC}^{\text{dyn}}} = \frac{t_{SA}^{\text{dyn}}}{t_{SA}^{\text{dyn}} + t_{TA}^{\text{dyn}}} = \frac{61}{61 + 98} = \frac{61}{159} \approx 0.38$$

Which means that 38% of the overall execution time is wasted in the scheduler. The scheduling overhead problem can be mended by coding the actors of the application at an appropriate level of granularity, that is, combining so much functionality into one actor that the overall computation time $t_{TA}$ dominates the overall scheduling time $t_{SA}$. Note that combining functionality from actors of very fine granularity might also reduce the overall computation time $t_{TA}$ required for a given piece of computational work. This is due to the fact that the compiler might be able to generate more efficient code if the functionality is combined in one function in contrast to a distributed implementation of the functionality amongst many fine-grained actors. Hence, there is a penalty for dynamic scheduling of fine-grained actors not only from the scheduling overhead but also from code generation deficiencies for the implemented functionality. Thus, the scheduling overhead alone might understate the problem.

To exemplify, consider the two variants of the network graph $g_{\text{sqr, flat}}$ (cf. Figure 4.1) shown in Figures 4.3 and 4.4. Both network graphs implement Newton’s
4. Clustering

iterative algorithm for calculating the square root. However, they differ in the number of actors that must be scheduled by a dynamic scheduler. In both variants, static actors have been shaded slightly gray. The \texttt{Src}, \texttt{Sink}, \texttt{Approx}, and \texttt{Dup} actors are marked graph actors consuming and producing one token each on their input and output ports in each of their firings. The communication behavior of the \texttt{SqrLoop} actor is given by the FSM depicted in Figure 3.5 on page 45. Moreover, in the network graph $g_{\text{sqr}}$ shown in Figure 4.4, the subcluster $g_{\text{gamma}}$ (cf. Figure 4.3) containing the actors $a_3$ and $a_4$ has been replaced by the composite actor $a_{\gamma_{\text{appx}}}$. This composite actor implements a QSS for the actors $a_3$ and $a_4$.

![Network graph $g_{\text{sqr}}$ and its subcluster $g_{\text{gamma}}$](image)

\textbf{Figure 4.3: Hierarchical network graph} containing the subcluster $g_{\text{gamma}}$

In order to represent a QSS, the notion of a so-called \textit{cluster FSM} is used. A cluster FSM resembles an actor FSM from Definition 3.2 on page 46 with the exception that the actions of the transitions execute sequences of statically scheduled actor firings. Intuitively, a composite actor is a cluster with a cluster FSM implementing a QSS for the actors of the cluster. In case of the composite actor $a_{\gamma_{\text{appx}}}$, the action $\langle a_3, a_4 \rangle \in a_{\gamma_{\text{appx}}}.\mathcal{F}_{\text{action}}$ of the transition $t_1$ implements the sequence $\langle a_3, a_4 \rangle$ of statically scheduled actor firings. More formally, a cluster FSM is defined as follows:

\textbf{Definition 4.1 (Cluster FSM [FKH+08*]).} The \textit{cluster FSM} $\mathcal{R}$ of a cluster $g_{\gamma}/$composite actor $a_{\gamma}$ is a tuple $(Q, q_0, T)$ containing a finite set of \textit{cluster states} $Q$, an \textit{initial cluster state} $q_0 \in Q$, and a finite set of \textit{transitions} $T$. A \textit{transition} $t \in T$ itself is a tuple $(g_{\text{src}}, k^{\text{io}}, f(\_), q_{\text{dest}})$ containing the source state
4.1 Clustering and Scheduling Overhead

Network graph $\tilde{g}_{\text{app}}$ and its composite actor $a_{\gamma_{\text{appx}}} \in \tilde{g}_{\text{app}}.A$

Figure 4.4: Refined network graph derived from replacing the $g_{\gamma_{\text{appx}}}$ subcluster by a corresponding composite actor $a_{\gamma_{\text{appx}}}$

$q_{\text{src}} \in Q$, from where the transition is originating, and the destination state $q_{\text{dst}} \in Q$, which will become the next current state after the execution of the transition starting from the current state $q_{\text{src}}$. Furthermore, if a transition $t$ is taken, then its action $f(\ldots) \in a_{\gamma}.\mathcal{F}_{\text{action}}$ implementing a sequence $(\ldots) \in g_{\gamma}.A^*$ of statically scheduled actor firings will be executed. Finally, the execution of a transition $t$ itself is guarded by an input/output guard $k^{io}_{\#c_2 \geq 1 \land \#c_3 \geq 1 \land \#c_4 \geq 1 \land \#c_5 \geq 1}$.

The replacement of the static cluster $g_{\gamma_{\text{appx}}}$ by the composite actor $a_{\gamma_{\text{appx}}}$ may reduce the overall scheduling time $t_{SA}$ as it condenses the execution times of the input/output guards $k^{io}_{\#c_2 \geq 1 \land \#c_3 \geq 1 \land \#c_4 \geq 1 \land \#c_5 \geq 1}$ and $k^{io}_{\#c_2 \geq 1 \land \#c_3 \geq 1 \land \#c_5 \geq 1}$ into one input/output guard $k^{io}_{\#c_2 \geq 1 \land \#c_3 \geq 1 \land \#c_4 \geq 1 \land \#c_5 \geq 1}$. This input/output guard requires less execution time than the sum of execution times required by the input/output guards $k^{io}_{\#c_2 \geq 1 \land \#c_3 \geq 1 \land \#c_4 \geq 1 \land \#c_5 \geq 1}$ and $k^{io}_{\#c_2 \geq 1 \land \#c_3 \geq 1 \land \#c_5 \geq 1}$. The input/output guard $k^{io}_{\#c_2 \geq 1 \land \#c_3 \geq 1 \land \#c_4 \geq 1 \land \#c_5 \geq 1}$ is used by the composite actor $a_{\gamma_{\text{appx}}}$ to guard the transition $t_1$. If this transition is taken, the action $f(\langle a_3,a_4 \rangle)$, which implements the sequence $\langle a_3,a_4 \rangle$ of statically scheduled actor firings, is executed. A possible schedule of the modified network graph $\tilde{g}_{\text{app}}$ is visualized by the Gantt chart shown in Figure 4.5. The scheduling overhead of this schedule can be computed as:

\[
\frac{t_{\text{EXEC}}^{\text{qss}}}{t_{\text{EXEC}}} = \frac{t_{\text{EXEC}}^{\text{qss}}}{t_{\text{EXEC}}^{\text{qss}} + t_{TA}^{\text{qss}}} = \frac{41}{41 + 77} = \frac{41}{118} \approx 0.35
\]

Moreover, the overall computation time $t_{TA}^{\text{qss}}$ has also been reduced from 98 to 77 abstract time units. This reduction can be explained by considering the more
efficient code generation and compiler optimizations possible for the sequence \((a_3, a_4)\) of statically scheduled actor firings as compared to compiler optimizations possible for the isolated firings of actor \(a_3\) and actor \(a_4\). To exemplify, in the actor functionality \(f_{(a_3, a_4)}\), the channel \(c_4\) can be replaced by a local variable, which could potentially be allocated into a CPU register by the compiler, while the channel \(c_3\) can be replaced by a global variable. Furthermore, read and write access to these two channels by the actor functionality of the actors \(a_3\) and \(a_4\) will be simplified accordingly.

Note that the timings used in the Gantt charts in this section have been derived from real measurements on the Xilinx Zynq-7000 platform. This platform contains an ARM® Cortex™-A9 dual-core CPU running at 667 MHz. First, the pure C++ software implementations have been generated by applying the synthesis back-end presented in Chapter 5 to the SysteMoC application implementing the square root DFG shown in Figure 4.3. For the refined DFG \(\tilde{g}_{\text{sqr}}\), the clustering methodology from Section 4.4.2 has been applied before code generation. Next, the executables have been generated by using version 4.7 of the gcc compiler suite with the -O3 option for the ARM® Cortex™-A9 target.
architecture. Finally, the measures have been taken on the Xilinx ZedBoard by using the measuring infrastructure included with the Linaro Linux-distribution.

To enable a better layout of these Gantt charts, the measurements have been converted to abstract time units by means of normalization and slight liberties have been taken with the individual execution times of the actions and guards. To exemplify, the real timings for a dynamically scheduled (cf. Figure 4.3) computation of 1,000,000 integer-based square root calculations are given in Equation (4.1). The corresponding timings for the refined DFG (cf. Figure 4.4) are given in Equation (4.2).

\[
\begin{align*}
\frac{t_{\text{dyn}}_{SA}}{t_{\text{EXEC}}^{\text{dyn}}} &= \frac{t_{\text{dyn}}_{SA}}{t_{\text{EXEC}}^{\text{dyn}}} = \frac{2.30 \text{ sec}}{2.30 \text{ sec} + 3.23 \text{ sec}} = \frac{2.30 \text{ sec}}{5.53 \text{ sec}} \approx 0.42 \quad (4.1) \\
\frac{t_{\text{qss}}_{SA}}{t_{\text{EXEC}}^{\text{qss}}} &= \frac{t_{\text{qss}}_{SA}}{t_{\text{EXEC}}^{\text{qss}}} = \frac{1.71 \text{ sec}}{1.71 \text{ sec} + 2.58 \text{ sec}} = \frac{1.71 \text{ sec}}{4.29 \text{ sec}} \approx 0.40 \quad (4.2)
\end{align*}
\]

The speedup \( s_{\text{qss}} \) between the dynamically scheduled implementation and the refined implementation is as follows:

\[
\frac{t_{\text{EXEC}}^{\text{dyn}}}{t_{\text{EXEC}}^{\text{qss}}} = \frac{5.53 \text{ sec}}{4.29 \text{ sec}} = s_{\text{qss}} \approx 1.29
\]

Note that this speedup cannot be explained alone from the schedule overhead reduction from \( t_{\text{dyn}}_{SA} = 2.30 \text{ sec} \) to \( t_{\text{qss}}_{SA} = 1.71 \text{ sec} \), but stems also from the improvement in code generation for the implementation of the functionality. That is to say, the overall computation times \( t_{\text{dyn}}^{\text{TA}} \) and \( t_{\text{qss}}^{\text{TA}} \) differ, yet they perform the same amount of useful work, i.e., 1,000,000 integer-based square root calculations. Hence, the scheduling overhead from Equation (4.1) understates the problem. To exemplify, a hand-optimized implementation of Newton’s iterative algorithm for calculating the square root is measured as \( t_{\text{EXEC}}^{\text{hand}} = 2.66 \text{ sec} \) for 1,000,000 integer-based square root calculations. Thus, the overall overhead for the two implementations derives from scheduling overhead and code generation deficiencies and can be calculated as follows:

\[
\begin{align*}
\frac{t_{\text{EXEC}}^{\text{dyn}} - t_{\text{EXEC}}^{\text{hand}}}{t_{\text{EXEC}}^{\text{EXEC}}} &= \frac{5.53 \text{ sec} - 2.66 \text{ sec}}{5.53 \text{ sec}} \approx 0.52 \\
\frac{t_{\text{EXEC}}^{\text{qss}} - t_{\text{EXEC}}^{\text{hand}}}{t_{\text{EXEC}}^{\text{EXEC}}} &= \frac{4.29 \text{ sec} - 2.66 \text{ sec}}{4.29 \text{ sec}} \approx 0.38
\end{align*}
\]

Note that this overall overhead is mathematically equivalent to the speedup if only singlecore implementations are considered. To exemplify, the overhead \( s_{\text{overhead}} \) is defined as the factor by which the hand-optimized implementation is faster than the fully dynamically scheduled implementation:

\[
s_{\text{overhead}} = \frac{t_{\text{EXEC}}^{\text{dyn}}}{t_{\text{EXEC}}^{\text{hand}}} = \frac{5.53 \text{ sec}}{2.66 \text{ sec}} \approx 2.08
\]
Then, the overall overhead for each implementation can be computed from the speedup $s$ of the implementation and the overhead $s_{\text{overhead}}$ as follows:

$$1 - \frac{s}{s_{\text{overhead}}} = 1 - \frac{t_{\text{EXEC}}^{\text{dyn}}}{t_{\text{EXEC}}^{\text{impl}}} = 1 - \frac{t_{\text{EXEC}}^{\text{hand}}}{t_{\text{EXEC}}^{\text{impl}}} = \frac{t_{\text{EXEC}}^{\text{impl}}}{t_{\text{EXEC}}^{\text{impl}}} - \frac{t_{\text{EXEC}}^{\text{hand}}}{t_{\text{EXEC}}^{\text{impl}}}$$

Hence, in Section 4.5, which presents results for singlecore implementations, only the speedup needs to be considered.

However, combining functionality of multiple actors into one actor can also mean that more data has to be produced and consumed atomically which requires larger channel capacities and may delay computation unnecessarily, therefore degrading the latency if multiple computation resources are available for execution of the application. Hence, selecting a level of granularity for a functionality by the designer is a manual trade-off between schedule efficiency to improve throughput and latency on a dedicated CPU solution against resource consumption, e.g., larger channel capacities, and scheduling flexibility required to improve throughput and latency on a multicore solution. Moreover, if latency minimization is a secondary goal as compared for example to energy minimization, then clustering might improve energy efficiency due to overhead minimization even if the overall latency is not improved. Thus, if the mapping of actors to resources itself is part of the synthesis step, e.g., as is the case in the SystemCoDesigner [HFK+07*, KSS+09*] DSE methodology, an appropriate level of granularity can no longer be chosen in advance by the designer as it depends on the mapping itself.

An example of such a situation is the network graph $g_{\text{ex}1}$ and its two possible implementations shown in Figure 4.6. Dynamic actors are represented by white rounded boxes, e.g., $a_5$, while static actors are represented by shaded rounded boxes, e.g., $a_1 - a_4, a_6$.

All static actors except actor $a_4$ are HSDF actors. In contrast, the actor $a_4$ consumes three tokens from channel $c_4$ and produces one token on channel $c_5$ when fired. The dynamic actor $a_5$ determines how often a data value should be processed by the actors $a_2 - a_4$. The two Gantt charts shown in Figure 4.7 depict possible schedules of the unmodified network graph $g_{\text{ex}1}$ for the two implementations from Figure 4.6. Each Gantt chart only shows a possible schedule of the graph up to the point in time at which the first result token arrives at actor $a_6$. The schedules depicted in the Gantt charts assume that each resource has its own dedicated dynamic scheduler as given in Algorithm 2.

As can be seen, the dual CPU implementation finishes the processing of the first result token by actor $a_6$ at 103 abstract time units, while the single CPU implementation requires 135 abstract time units. In order to improve this latency, QSS is now applied. However, as can be seen from Figures 4.8 and 4.9,
different QSSs are required in order to obtain a smaller latency for the different implementations. Two clustering alternatives are considered: (1) clustering and refining the actors $a_2$ and $a_3$ into a composite actor $a_{72,3}$ (cf. Figure 4.9), and (2) further enlarging the cluster by adding the actor $a_4$ to derive the composite actor $a_{72,3,4}$ (cf. Figure 4.8). It turns out that the second alternative with the composite actor $a_{72,3,4}$ produces a smaller latency (91 instead of 114 abstract time units) for the single CPU implementation.

However, the sequence $\langle a_2, a_3, a_2, a_3, a_2, a_3, a_4 \rangle$ of statically scheduled actor firings realized by the action $f(\langle a_2, a_3, a_2, a_3, a_3, a_4 \rangle)$ of the composite actor $a_{72,3,4}$ can only be executed when at least three tokens are available in the channel $c_2$.

Hence, for the second alternative, the channel capacity size($c_2$) of the channel $c_2$ must be able to accommodate at least three tokens. On the other hand, for the first alternative and the unclustered case, a channel capacity of one token is sufficient. Indeed, if no information is known about actor $a_5$, then the channel capacity has to be increased from one to five tokens in order to compensate for the inaccessible channel capacities of size($c_3$) = 1 and size($c_4$) = 3 tokens of the channels $c_3$ and $c_4$. A detailed treatment of the problem is given in Section 4.4.6.

Furthermore, the requirement for at least three tokens in order to execute $f(\langle a_2, a_3, a_2, a_3, a_2, a_3, a_4 \rangle)$ will delay the computation of the first actor firing of actor $a_2$ compared to the schedules depicted in Figure 4.7. This does not harm the latency in the single CPU implementation (cf. Figure 4.8a) because the CPU $r_1$ can perform other useful work until the action $f(\langle a_2, a_3, a_2, a_3, a_3, a_4 \rangle)$ can be executed. However, in the dual CPU implementation (cf. Figure 4.8b), the only actor bound onto the second CPU $r_2$ is the composite actor $a_{72,3,4}$. Hence, the first 13 schedule rounds $I-XIII$ of $r_2$ only consist of scheduling overhead. The
4. Clustering

(a) Gantt chart of a possible schedule for a single CPU implementation of the graph $g_{ex_1}$

(b) Gantt chart of a possible schedule for a dual CPU implementation of the graph $g_{ex_1}$

**Figure 4.7**: Gantt charts for the unmodified network graph $g_{ex_1}$

first clustering alternative with the composite actor $a_{7,3}$ does not have this problem. Hence, for the dual CPU implementation, this alternative (cf. Figure 4.9b) produces a smaller latency (75 instead of 82 abstract time units).

To overcome these challenges, the presented clustering-based design flow allows the DSE to decide the set of actors that should be clustered. In the following, however, a heuristics will be used that maximizes the size of the created clusters.
4.2 Clustering as an Operation of Refinement

Clustering refinement is the conversion of a cluster $g_\gamma$ into a single composite actor $a_\gamma$ of equivalent behavior. Hence, a notion for equivalent behavior is required. A static cluster has sequence determinate behavior [LSV98], i.e., the behavior of the cluster is independent of the scheduling of the actors inside the cluster. Thus, it can be modeled by a Kahn function via the relation as presented in [Lee97] between Kahn’s denotational semantics [Kah74] and the semantics of data flow models with the notion of firing.\footnote{The requirement for sequence determinate behavior is also satisfied if the cluster contains Kahn actors. Therefore, the equivalence condition as introduced in Definition 4.2 could be reused in the context of a more general refinement of clusters containing Kahn actors.} However, to use this denotational
4. Clustering

With the infinite channel capacities simplification, the denotational semantics, infinite channel sizes for the channels connected to the composite actor ports have to be assumed. In case of cluster refinement in static DFGs as presented in Section 4.3, this restriction does not matter as determining the channel capacities at compile time in order to satisfy various criteria are decidable problems [SGB06a, SGB08, BMMKU10, BMMKM10, WBJS06, WBS07] for static DFGs.

However, for dynamic DFGs, even the question if the graph can be executed in bounded memory is undecidable [Buc93]. Hence, in the following, the refinement algorithms presented in the Sections 4.3.1, 4.3.2, 4.4.2, and 4.4.3 as well as the correctness proof in Section 4.4.5 will assume infinite channel capacities. The problem of finite channel capacities is discussed in Section 4.4.6.

With the infinite channel capacities simplification, the denotational semantics of Kahn Process Networks (KPNs) can be used to compare the cluster and its

---

**Figure 4.9:** Gantt charts for the clustered network graph that results from clustering and refining the actors $a_2$ and $a_3$ into a composite actor $a_{\gamma_{2,3}}$
corresponding composite actor for equivalence. More formally, the equivalence between a cluster and its corresponding composite actor is defined as follows:

**Definition 4.2** (Equivalence Condition for Sequence Determinate Clusters). Given a sequence determinate cluster \( g_\gamma \) and a composite actor \( a_\gamma \) as well as their corresponding Kahn functions \( K_{g_\gamma} \) and \( K_{a_\gamma} \), then the cluster \( g_\gamma \) and the composite actor \( a_\gamma \) have equivalent behavior if and only if \( K_{g_\gamma} \equiv K_{a_\gamma} \) and the cluster/composite actor is connected to its cluster/composite actor environment with channels of infinite size.

The equivalence condition ensures two different things: (1) It requires the cluster and the composite actor to produce the same sequences of values on output ports when the cluster and the composite actor are given the same infinite sequences of input values on their input ports, and (2) it requires the cluster and the composite actor to have the same data dependencies between the consumed tokens on the input ports and the produced tokens on the output ports.

The satisfaction of requirement (1) by a cluster refinement operation is relatively straightforward. It merely requires the composite actor and the cluster to perform the same operations on the token values. Thus, in the following only the satisfaction of requirement (2) is considered. Requirement (2) prevents the possible introduction of an artificial deadlock by the cluster refinement operation due to missing tokens.\(^{33}\)

The key idea of Definition 4.2 is to satisfy the so-called worst case environment of the cluster since satisfying this environment will also preserve deadlock freedom for all other possible cluster environments. The worst case environment contains for each output port \( o \in g_\gamma.O \) and each input port \( i \in g_\gamma.I \) a feedback loop where any produced token on the output port \( o \) is needed for the activation of an actor \( a \in g_\gamma.A \) connected to the input port \( i \). To exemplify, the DFG \( g_{\text{sqrt}} \) and its subcluster \( g_{\text{sqrt appx}} \) depicted in Figure 4.3 on page 90 is considered. In this case, each token produced on the output port \( o_1 \in g_{\text{sqrt appx}}.O \) of the subcluster is required by the SqrLoop actor \( a_2 \) to produce the next token for consumption on the input port \( i_1 \in g_{\text{sqrt appx}}.I \) of the subcluster.

In particular, postponing the production of an output token would result in a deadlock of the entire system. To satisfy the equivalence condition, the composite actor produced by a cluster refinement must, thus, always produce the maximum number of output tokens possible from the consumption of a minimum number of input tokens.

A cluster refinement operator \( \xi \) is used to denote the refinement of a cluster into a composite actor. This operator transforms an original graph \( g \in G \) and

\(^{33}\)The deadlock is artificial in the sense that it was not present in the original DFG which was refined.
4. Clustering

its subcluster $g_\gamma \in g.G$ into a refined graph $\tilde{g}$ containing the actor $a_\gamma \in \tilde{g}.A$ replacing the subcluster while keeping the behavior of the original graph. This operation is exemplified in Figure 4.3 and Figure 4.4: Depicted is the original network graph $g_{\text{sqr}}$ and its static subcluster $g_{\text{\gamma appx}}$ which is transformed by the cluster refinement operation $\xi(g_{\text{sqr}}, g_{\text{\gamma appx}}) = (\tilde{g}_{\text{sqr}}, a_{\gamma \text{appx}})$ into the resulting refined network graph $\tilde{g}_{\text{sqr}}$ and the corresponding composite actor $a_{\gamma \text{appx}}$ derived from $g_{\gamma \text{appx}}$. More formally, the cluster refinement operation is defined as follows:

**Definition 4.3** (Cluster Refinement Operation). Given a graph $g$ and its sequence determinate subcluster $g_\gamma$. The cluster refinement operator $\xi : G \times G \to G \times A$ replaces $g_\gamma$ by a single Kahn actor $a_\gamma$, called composite actor, resulting in a refined graph $\tilde{g}$, i.e., $\xi(g, g_\gamma) = (\tilde{g}, a_\gamma)$ where $g_\gamma \in g.G$ and $\tilde{g} = g$ except that $\tilde{g}.V = g.V + \{a_\gamma\} - \{g_\gamma\}$ as well as $a_\gamma.P = g_\gamma.P$.

If the subcluster $g_\gamma$ and the produced composite actor $a_\gamma$ satisfy the equivalence condition from Definition 4.2, then the cluster refinement operation will in the following be called a safe cluster refinement operation. Otherwise, the operation will be called unsafe. Furthermore, if the refined DFG is free of deadlocks, then the refinement will be called a valid cluster refinement operation. Otherwise, the refinement operation will be called invalid.

Note that safety is a local property that is decidable from the static cluster and the produced composite actor, while validity is a global property that is decidable from a composite actor and the static DFG containing this composite actor. Moreover, validity in general is undecidable if the composite actor is contained in a dynamic DFG. In contrast, if the original (dynamic) DFG is free of deadlocks and the cluster refinement operation is a safe cluster refinement, then the cluster refinement operation is also valid and will produce a refined DFG that is also free of deadlocks. The above given fact is a corollary of process composition as defined by Lee et al. in [LSV98] and the equivalence of $K_{g_\gamma}$ and $K_{a_\gamma}$ as required by Definition 4.2 and proven in Section 4.4.5. Thus, the safety property of a refinement operation can be used as a fallback for refinements of static clusters in dynamic DFGs.

4.3 Cluster Refinement in Static Data Flow Graphs

In the following, a refinement of a static subcluster $g_\gamma \in g_n.G$ of a network graph $g_n$ only containing actors exhibiting SDF or CSDF semantics is considered. It will be assumed that the network graph $g_n$ is a consistent and deadlock-free SDF or CSDF graph (cf. Section 2.2 on page 9).

Various forms of representing the composite actor $a_\gamma$ are compared for their feasibility to represent the result of a safe cluster refinement operation. Con-
4.3 Cluster Refinement in Static Data Flow Graphs

4.3.1 Refining a Cluster to an SDF Actor

Refining a cluster to an SDF composite actor, which implements a static schedule executing one iteration of the SDF or CSDF graph contained in the cluster, is called monolithic clustering. This approach excels at schedule overhead reduction. An example of such a best case scenario is shown in Figure 4.10. In this scenario, the static subcluster for the 2-dimensional IDCT can be replaced by an SDF composite actor. The SDF composite actor implements a static schedule of the contained IDCT actors, i.e., only the input/output guard $A_{i_1 \geq 64 \land o_1 \geq 64}$ needs to be satisfied, before the composite actor can fire, thus, executing the calculated static schedule. However, as will be later shown, this approach might be infeasible in the general case due to the cluster environment.

More formally, the $\xi_{SDF}$ operator will be used to denote the refinement operator producing SDF composite actors. To exemplify, the SDF graph depicted in

---

$34$ In general, the longer the static scheduling sequence which can be executed by checking a single input/output guard, the less schedule overhead is imposed by this schedule.
Figure 4.11a is considered. A firing of the resulting SDF composite actor $a_{\gamma_{1,2}}$ depicted in Figure 4.11b executes the sequence $\langle a_1, a_2, a_2 \rangle$ of statically scheduled actor firings. For code generation, efficiently looped versions of this sequence of actor firings, e.g., fire actor $a_1$ once then loop actor $a_2$ twice, can be determined by applying the scheduling algorithms presented in [BL93, BBHL95, BML97] to the repetition vector \[ \eta_{\gamma_{1,2}} = (\eta_1^{\text{rep}}, \eta_2^{\text{rep}}) = (1, 2) \] for the subcluster $g_{\gamma_{1,2}}$ derived via the cluster operation \[ \Gamma(g_{\text{ex}_{2}}, \{ a_1, a_2 \}) = (\tilde{g}_{\text{ex}_{2}}, g_{\gamma_{1,2}}). \]

![Diagram](image)

(a) A simple SDF graph $g_{\text{ex}_{2}}$ containing the two actors $a_1$ and $a_2$

(b) Clustered and refined DFG $\tilde{g}_{\text{ex}_{2}}$ containing the SDF composite actor $a_{\gamma_{1,2}}$

**Figure 4.11:** Example for a safe and valid refinement of the two SDF actors $a_1$ and $a_2$ into a composite SDF actor $a_{\gamma_{1,2}}$

However, while the cluster refinement $\xi_{\text{SDF}}(g'_{\text{ex}_{2}}, g'_{\gamma_{1,2}}) = (\tilde{g}_{\text{ex}_{2}}, a_{\gamma_{1,2}})$ depicted in Figure 4.11 is a safe cluster refinement operation, i.e., it will not introduce deadlock into the system regardless of the cluster environment connected to the subcluster, the $\xi_{\text{SDF}}$ operator performs an unsafe cluster refinement operation in the general case. Hence, for some combinations of cluster and cluster environment, the refinement via the $\xi_{\text{SDF}}$ operator might introduce a deadlock.

To exemplify, the cluster refinement for a cluster containing the two actors $a_3$ and $a_4$ is performed for the two different DFGs $g_{\text{ex}_{3a}}$ and $g_{\text{ex}_{3b}}$ that are shown in Figures 4.12 and 4.13, respectively. As can be seen in Figure 4.12, the $\xi_{\text{SDF}}$ operation in general is unsafe due to the introduction of additional constraints into the scheduling of the resulting DFG. These constraints may introduce an artificial deadlock. To exemplify, the composite actor $a_{\gamma_{3,4}}$ produced by the cluster refinement operation imposes the constraint that the actor $a_4$ must fire first followed by three firings of the actor $a_3$. Hence, neither the actor $a_3$ contained in the composite actor $a_{\gamma_{3,4}}$ nor the composite actor $a_{\gamma_{3,4}}$ itself, nor the sole actor $a_{\text{env}}$ in the cluster environment can fire. Thus, the refinement operation does produce a deadlock for the resulting DFG $\tilde{g}_{\text{ex}_{3a}}$ depicted in Figure 4.12b.

This is due to the fact that the two initial tokens on the channel between actor $a_4$ and actor $a_3$ in the original graph depicted in Figure 4.12a are no longer accessible in the refined graph $\tilde{g}_{\text{ex}_{3a}}$. Adding further initial tokens to the cluster environment, as has been done in the DFG $g_{\text{ex}_{3b}}$ depicted in Figure 4.13,
4.3 Cluster Refinement in Static Data Flow Graphs

Figure 4.12: Example for an unsafe and invalid refinement of the two SDF actors $a_3$ and $a_4$ by the $\xi_{\text{SDF}}$ operator

(resolves this problem. Hence, the refined DFG $\tilde{g}_{\text{ex3b}}$ depicted in Figure 4.13b is free of deadlocks.

Figure 4.13: Example for an unsafe but valid refinement of the two SDF actors $a_3$ and $a_4$ by the $\xi_{\text{SDF}}$ operator

These situations can be analyzed in detail (cf. Figure 4.14) with the help of a so-called Acyclic Precedence Graph (APG) [LM87a, SB00]. The APG can be derived from the unfolding of the SDF or CSDF graph into an equivalent marked graph and the subsequent deletion of all edges of the marked graph which are carrying initial tokens. In the following figures, marked graphs are depicted in order to also show the edges with the initial tokens which would have been deleted if the corresponding APG was shown. The unfolding operation generates one marked graph actor, a vertex in the unfolded graph, for each firing of the corresponding SDF actor. To exemplify, the SDF graph $g_{\text{ex2}}$ depicted in Figure 4.11a and its marked graph unfolding as shown in Figure 4.14a are considered. The graph $g_{\text{ex2}}$ has a repetition vector of $\eta_{\text{rep}} = (\eta_1^{\text{rep}}, \eta_2^{\text{rep}}, \eta_{\text{env}}^{\text{rep}}) = (3, 6, 2)$. Hence, the marked graph derived by unfolding $g_{\text{ex2}}$ has three vertices $a_1^2$, $a_1^3$, and $a_1^4$ corresponding to the three firings of the SDF actor $a_1$. Therefore, the unfolding operation presented in [LM87a, SB00] are in general of exponential complexity in the size of the problem description of the original SDF graph.
4. Clustering

Figure 4.14: Depicted above are the marked graph expansion of the DFG $g_{ex_2}$ depicted in Figure 4.11a and the corresponding coalesced vertices depicted as dark shaded areas for the refinement of the cluster $g_{\gamma_{1,2}}$ into the composite actor $a_{\gamma_{1,2}}$.

The channels, the edges in the unfolded graph, are generated in such a way by the unfolding operation that the data dependencies between the actor firings in the original SDF graph are preserved. The interested reader is referred to the original literature [LM87a, SB00] for a detailed description of the unfolding operation.

As presented in Section 2.2.1 on page 10, a marked graph has a deadlock if and only if there exists a directed cycle in the marked graph where none of the channels, the edges in the marked graph, in the cycle carries any initial tokens. Hence, a directed cycle in the APG signifies a deadlock in the underlying SDF or marked graph. Now, refinements can be represented in the APG by coalescing actor firings (vertices) in the APG and only refinements of subclusters into composite actors which do not generate cycles by coalescing the corresponding actor firings of the refinement in the APG may be shown to produce valid transformations.

To exemplify, the cluster refinement $(\tilde{g}_{ex_2}, a_{\gamma_{1,2}}) = \xi_{SDF}(g'_{ex_2}, g_{\gamma_{1,2}})$ is considered. To recapitulate, the repetition vector of the SDF graph $g_{ex_2}$ is $\eta_{ex_2}^{rep} = (\eta_1^{rep}, \eta_2^{rep}, \eta_{env}^{rep}) = (3, 6, 2)$ while the subcluster $g_{\gamma_{1,2}}$ has a repetition vector of $\eta_{\gamma_{1,2}}^{rep} = (\eta_1^{rep}, \eta_2^{rep}, \eta_{env}^{rep}) = (3, 6, 2)$.

While more advanced techniques for unfolding exist [Gei09], they generate a marked graph which is conservative for the purpose of throughput analysis and in general contains additional dependencies. Hence, the generated marked graph is unusable for the purpose of determining the validity of a cluster refinement.
4.3 Cluster Refinement in Static Data Flow Graphs

Figure 4.15: Shown above are the marked graph expansions as well as the coalesced vertices depicted as dark shaded areas for the DFGs \( g_{ex3a} \) and \( g_{ex3b} \). Edges which are part of dependency cycles are highlighted in bold.

\[ \eta_{rep} = (\eta_1^{rep}, \eta_2^{rep}) = (1, 2) \]. Hence, in the marked graph unfolding of \( g_{ex2} \), the refinement operation will lead to a coalescing of one actor firing, as \( \eta_1^{rep} = 1 \), of the SDF actor \( a_1 \) and two actor firings, as \( \eta_2^{rep} = 2 \), of the SDF actor \( a_2 \) into a new marked graph actor firing. In total, there will be three such coalescings in the marked graph as determined from the equation below:

\[ \frac{\eta_1^{rep}}{\eta_1^{rep'}} = \frac{\eta_2^{rep}}{\eta_2^{rep'}} = 3 \]

This can be observed by considering Figure 4.14b where the actor firing \( a_1^1, a_2^2 \), and \( a_2^3 \) have been coalesced into the new composite actor firing \( a_{7,1,2}^1 \), depicted as the first coalescing (dark shaded area) of the three coalescings in Figure 4.14b. In the following, the vertices that are coalesced, e.g., the vertices \( a_1^1, a_2^3, \) and \( a_2^2 \), and the edges connecting these vertices, e.g., the edges \( a_1^1 \rightarrow a_2^2 \) and \( a_1^1 \rightarrow a_2^3 \), will be called the coalescing. The composite actor firings \( a_{7,1,2}^1 \) and \( a_{7,1,2}^3 \) have been built accordingly from the constituent actor firings of the actors \( a_1 \) and \( a_2 \).

The three situations encountered in the cluster refinements depicted in Figures 4.11 to 4.13 are now analyzed in detail via usage of the APGs depicted in Figures 4.14 and 4.15. These situations correspond to the three cases as given below:
The refinement is a safe cluster refinement. This case corresponds to the subcluster refinement \( (\tilde{g}_{\text{ex}2}, a_{\gamma_{1,2}}) = \xi_{\text{SDF}} \circ \Gamma(g_{\text{ex}2}, \{a_1, a_2\}) \) depicted in Figure 4.11 with the corresponding APG shown in Figure 4.14b.\(^{37}\)

Note that a safe cluster refinement operation, that is an operation conforming to the equivalence condition given in Definition 4.2, does not guarantee that the resulting refined DFG is free of deadlocks. It only ensures that no additional artificial deadlocks are introduced by the refinement operation. To exemplify, if the DFG \( g_{\text{ex}2} \) is modified by removing the two initial tokens from the channel from \( a_{\text{env}} \) to \( a_1 \), then the cluster refinement operation \( (\tilde{g}_{\text{noini}2}, a_{\gamma_{1,2}}) = \xi_{\text{SDF}} \circ \Gamma({g}_{\text{noini}2}, \{a_1, a_2\}) \) is still a safe cluster refinement operation. However, both the original DFG \( g_{\text{noini}2} \) as well as the resulting refined DFG \( \tilde{g}_{\text{noini}2} \) exhibit a deadlock.

In the APG, the equivalence condition translates to ensuring that the coalescing operations will not introduce additional cycles into the APG by replacing each coalescing and its contained actor firings with a composite actor firing.\(^{38}\) This can be ensured by checking the condition given below:

**Theorem 4.1 (Safe SDF Cluster Refinement).** Given a consistent and deadlock-free static subcluster \( g_{\gamma} \) of a DFG \( g \) and the APG expansion of the subcluster \( g_{\gamma} \), that is the coalescing of actor firings corresponding to this refinement operation. Then, the refinement operation \( \xi_{\text{SDF}}(g, g_{\gamma}) \) is a safe cluster refinement operation if and only if for each pair \((a_{\text{in}}^x, a_{\text{out}}^y)\) of input/output actor firings of the coalescing, there exists a directed path in the coalescing from the input actor firing \( a_{\text{in}}^x \) to the output actor firing \( a_{\text{out}}^y \).\(^{39}\)

**Proof.** The key idea of Theorem 4.1 is that the SDF refinement operation must not introduce dependencies on tokens consumed by the composite actor for tokens produced by the composite actor that were not already present in the original cluster. The SDF refinement operation introduces for all output actor firings in the APG expansion of the subcluster dependencies on all input actor firings in the APG expansion of the subcluster. Hence, these dependencies must already be present in the APG expansion of the subcluster itself. Thus, there must exist at least one directed path

\(^{37}\)The ‘\(\circ\)’-operator, e.g., \(\xi_{\text{SDF}} \circ \Gamma\), is used to denote function composition, i.e., \( (\xi_{\text{SDF}} \circ \Gamma)(g, \{a_n \ldots a_m\}) = \xi_{\text{SDF}}(\Gamma(g, \{a_n \ldots a_m\})) \).

\(^{38}\)There may already be cycles present in the APG. This is the case when the original DFG, e.g., \( g_{\text{noini}2} \), has a deadlock.

\(^{39}\)The directed path must consist of APG edges contained in the coalescing. Hence, as the figures are all showing the marked graph expansion only edges carrying no initial tokens must be used to form the directed path.
4.3 Cluster Refinement in Static Data Flow Graphs

in the original APG expansion of the subcluster $g_\gamma$ for each combination of input actor firing and output actor firing.

To exemplify, the APG expansion of the subcluster $g_{\gamma,1,2}$ of the DFG $g_{\text{ex}}$ shown in Figure 4.11a appears three times in the APG expansion (cf. Figure 4.14b) of the DFG $g_{\text{ex}}$ itself. These three occurrences (shaded areas) correspond to the three firings of the composite actor $a_{\gamma,1,2}$. Only one of the three coalescings needs to be checked as they are all equivalent. Here, the coalescing $a_{\gamma,1,2}^1$ is considered. The set of input actors of the subcluster $g_{\gamma,1,2}$ is $A_I = \{ a_1 \}$ while the set of output actors is $A_O = \{ a_2 \}$. Hence, the set of pairs of input/output actor firings is $\{(a_1^1, a_1^2), (a_1^1, a_2^2)\}$ and the directed path for each pair from the input actor firing to the output actor firing consists of the sole edge connecting both actors in the pair.

- The refinement is an unsafe cluster refinement and introduces an artificial deadlock. This case corresponds to the subcluster refinement $(\tilde{g}_{\text{ex}a}, a_{\gamma,3,4}) = \xi_{\text{SDF}} \circ \Gamma(g_{\text{ex}a}, \{ a_3, a_4 \})$ depicted in Figure 4.12 with the corresponding APG shown in Figure 4.15a.

One can see that the cluster refinement introduces a deadlock by observing the dependency cycles $a_{\gamma,3,4}^1 \rightarrow a_{\text{env}}^1 \rightarrow a_{\gamma,3,4}^1$ and $a_{\gamma,3,4}^2 \rightarrow a_{\text{env}}^2 \rightarrow a_{\gamma,3,4}^2$ in the APG depicted in Figure 4.15a. The first cycle $a_{\gamma,3,4}^1 \rightarrow a_{\text{env}}^1 \rightarrow a_{\gamma,3,4}^1$ is introduced because the coalescing $a_{\gamma,3,4}^1$ does not have an initial token free directed path from the input actor firing $a_1^1$ to the output actor firings $a_3$ and $a_3^2$. The second cycle $a_{\gamma,3,4}^2 \rightarrow a_{\text{env}}^2 \rightarrow a_{\gamma,3,4}^2$ is introduced because the coalescing $a_{\gamma,3,4}^2$ does not have an initial token free directed path from the input actor firing $a_1^2$ to the output actor firings $a_3$ and $a_3^5$. Hence, the cluster refinement $\xi_{\text{SDF}}(g_{\text{ex}a}, \{ a_3, a_4 \})$ adds new constraints in the form of artificial data dependencies between the pairs $(a_1^1, a_1^2), (a_1^1, a_2^2), (a_2^4, a_2^5)$, and $(a_2^2, a_3^5)$ of input/output actor firings.

These artificial data dependencies together with the data dependencies in the cluster environment form the two cycles in the APG which signify the introduced artificial deadlock. However, as can be seen in the next case, if the cluster environment is benign, then even the introduction of these artificial data dependencies may not introduce a deadlock.

- The refinement is an unsafe cluster refinement but results in no deadlock. This case corresponds to the subcluster refinement $(\tilde{g}_{\text{ex}b}, a_{\gamma,3,4}) = \xi_{\text{SDF}} \circ \Gamma(g_{\text{ex}b}, \{ a_3, a_4 \})$ depicted in Figure 4.13 with the corresponding APG shown in Figure 4.15b.

Compared to the previous case, the cluster environment is more forgiving. Hence, the introduction of the artificial data dependencies between the
pairs \((a_1^1, a_3^1)\) and \((a_1^2, a_3^2)\) do not cause a cycle in the APG as the edges \(a_3^1 \rightarrow a_{env}^1\) and \(a_3^2 \rightarrow a_{env}^1\) are missing in this environment. This global analysis approach to clustering is employed in [BL93].

In some cases, retiming [LLW07] can be used to enable a safe SDF refinement operation. In brief, retiming is an operation firing actors of the DFG in order to achieve a desired redistribution of the initial tokens.

Figure 4.16: Example to demonstrate that retiming can be used in some cases to make \(\xi_{SDF}\) a safe cluster refinement operation

To exemplify, the retimed DFG depicted in Figure 4.16 derived from the original DFG shown in Figure 4.12 is considered. The retiming is achieved by firing actor \(a_3\) twice, thus, moving the two initial tokens on the channel between actor \(a_4\) and \(a_3\) in the original graph depicted in Figure 4.12a outside of the subcluster. Hence, the problem of hiding initial tokens of the subcluster by the refinement operation \(\xi_{SDF}\) is averted.

As can be seen in the APG depicted in Figure 4.16c, retiming translates the unsafe cluster refinement operation \(\xi_{SDF} \circ \Gamma(g_{exa}, \{a_3, a_4\})\) into the safe cluster
4.3 Cluster Refinement in Static Data Flow Graphs

refinement operation $\xi_{SDF} \circ \Gamma(g_{ex_3}, \{ a_3, a_4 \})$. In detail, it can be seen in the APG that for both coalescings and for all their pairs of input/output actor firings $\{(a'_1, a'_2), (a'_3, a'_4), (a'_5, a'_6)\}$ and $\{(a^3_1, a^3_2), (a^3_3, a^3_4), (a^3_5, a^3_6)\}$, there exists a directed path from the input actor firing to the output actor firing containing only edges of the coalescing carrying no initial tokens.

However, retiming at compile time can only be employed if the behavior of the actors which are fired during the retiming operation are exactly known by the compiler and are free of side effects on parts of the system not under the control of the compiler. Otherwise, the behavior of the actors fired during the retiming operation cannot be replicated at compile time and can only be executed at run time. This compile-time retiming problem can be solved by generating a composite actor executing the retiming operation at run time. To exemplify, the composite actor as depicted in Figure 4.16d is considered. The retiming operation is encoded in the transition $t_1$ while the normal SDF composite actor mode corresponds to the self-loop via transition $t_2$.

Generally, however, a refinement of a subcluster to a composite SDF actor is an unsafe cluster refinement operation. Hence, it may introduce artificial deadlocks in the general case. This is true even in the presence of retiming transformations. To exemplify, the SDF graph $g_{ex_4}$ and its retimed variant $g_{r ex_4}$ depicted in Figures 4.17 and 4.18 are considered.

(a) SDF graph $g_{ex_4}$ with subcluster $g_{\gamma_5,6}$

(b) Resulting deadlocked DFG $\overline{g_{ex_4}}$

Figure 4.17: Example for an invalid refinement of the subcluster $g_{\gamma_5,6}$ into the SDF composite actor $a_{\gamma_5,6}$

While retiming can remove the hidden token problem of the two initial tokens carried on the edge from actor $a_6$ to actor $a_5$, the deadlock induced by the SDF refinement constraint that two firings $\eta^{\text{rep}}_{\gamma_{5,6}}(a_6) = 2$\footnote{Please remember, a vector, e.g., $\eta^{\text{rep}}_{\gamma_{5,6}} = (\eta^{\text{rep}}_{a_5}, \eta^{\text{rep}}_{a_6})$, is always also considered to be a function from its index set, e.g., here $\mathcal{I}(\eta^{\text{rep}}_{\gamma_{5,6}}) = \{a_5, a_6\}$, to its entries, e.g., $\eta^{\text{rep}}_{\gamma_{5,6}}(a_6) = \eta^{\text{rep}}_{a_6}$. Furthermore, to avoid the usage of double or triple subscripts, the notation $\eta^{\text{rep}}_{a}$ instead of $\eta^{\text{rep}}_{a_a}$ and similar constructs is given preference.} of actor $a_6$ have to be
4. Clustering

Figure 4.18: Example to demonstrate that retiming can not always be used to make $\xi_{SDF}$ a safe cluster refinement operation

4.3.2 Refining a Cluster to a CSDF Actor

In the following, the possibility of refining a subcluster into a CSDF composite actor is explored. More formally, the $\xi_{CSDF}$ operator will be used to denote a refinement operator producing a CSDF composite actor. Note that in contrast to the $\xi_{SDF}$ operator, there may exist multiple operators producing different CSDF composite actors from one subcluster. Here, CSDF composite actors will be called different if they differ in their communication behavior, that is taking a black box view they differ in the observable behavior from the outside.\footnote{If a white box view is taken, then there may even exists multiple versions of SDF refinement operators, each of which produce a different static schedule of the SDF actors contained in the composite actor.}

It will be shown that a refinement into a CSDF composite actor is sufficient to guarantee a valid cluster refinement operation for all possible subclusters if the cluster environment is known, consists only of SDF and CSDF actors, and the original DFG is consistent and free of deadlocks. However, a refinement into a CSDF composite actor is insufficient to guarantee a safe cluster refinement operation for an arbitrary static cluster. That is to say, a refinement into a CSDF composite actor is insufficient to prevent deadlock for all possible cluster environments of the arbitrary static cluster even if the cluster environment contains only SDF or CSDF actors.\footnote{This case corresponds to the modular code generation scenario explored in [TBG+13].}

To exemplify, refinements of the subclusters $g_{\gamma_{5,6}}$ and $g_{\gamma_{6,6}}$ of the SDF graphs depicted in Figures 4.17 and 4.18 into a CSDF composite actor each are now...
4.3 Cluster Refinement in Static Data Flow Graphs

considered. The CSDF refinement enables the resolution of the deadlock situation caused by the atomic firings of the sequence \(a_6, a_6, a_5\) required by a refinement to an SDF composite actor.

The situation for the SDF graph \(g_{ex4}\) can be analyzed in detail via usage of the marked graph expansion as depicted in Figure 4.19. The artificial deadlock introduced by the refinement operation \((g_{ex4}, a_{\gamma,5,6}) = \xi_{SDF}(g_{ex4}, g_{\gamma,5,6})\) can arise due to the violation of the safe SDF cluster refinement condition from Theorem 4.1 for the coalescing of actor firings from Figure 4.19b. From the four pairs \((a_1, a_2), (a_1, a_3), (a_2, a_1), (a_2, a_3)\) of input/output actor firings, only the pair \((a_1, a_2)\) does not violate the condition. To avoid these violations, the coalescing of actor firings can be split as depicted in Figure 4.19c. However, this changes the semantics of the composite actor \(a_{\gamma,5,6}'\) to be CSDF instead of SDF.

![Figure 4.19](image)

**Figure 4.19:** Depicted is a marked graph expansion of the subcluster \(g_{\gamma,5,6}\) as well as a coalescing and grouping of actor firings representing refinements of the subcluster to the SDF composite actor \(a_{\gamma,5,6}\) and the CSDF composite actor \(a_{\gamma,5,6}'\), respectively.

Each coalescing, depicted as a dark shaded area, of a grouping of actor firings corresponds to a CSDF phase. There are three coalescings corresponding to the three CSDF phases of the composite actor \(a_{\gamma,5,6}'\) depicted in Figure 4.20.

The consumption and production rates of the CSDF phases are annotated to the visual depiction of a CSDF actor as introduced in Section 2.2.3. Furthermore, due to lack of explicit channel annotations in the figures, the notation \(\text{cons}(a_1, a_2)/\text{prod}(a_1, a_2)\) is used to denote the SDF or CSDF consumption/production rate(s) of the actor \(a_2/a_1\) on the channel between the actors \(a_1\) and \(a_2\).
4. Clustering

Figure 4.20: Refinement \((g_{ex4}^{\gamma}, a_{\gamma_{5,6}}') = \xi_{CSDF}(g_{ex4}, g_{\gamma_{5,6}})\) of the subcluster \(g_{\gamma_{5,6}}\) of the graph \(g_{ex4}\) into the CSDF composite actor \(a_{\gamma_{5,6}}'\)

More formally, the consumption and production rates of the three CSDF phases of the composite actor \(a_{\gamma_{5,6}}'\) are given as follows:

\[
\text{prod}(a_{\gamma_{5,6}}', a_{env_1}) = (2, 0, 0)
\]
\[
\text{cons}(a_{env_2}, a_{\gamma_{5,6}}') = (0, 1, 1)
\]
\[
\text{prod}(a_{\gamma_{5,6}}', a_{env_2}) = (0, 1, 1)
\]

The consumption and production rate of the first CSDF phase is derived from the first coalescing \(a_{\gamma_{5,6}}'\) by noting that the firing \(a_{\gamma_{5,6}}'\) produces two tokens \((\text{prod}(a_{\gamma_{5,6}}', a_{env_1}))(1) = 2\) on the channel from actor \(a_{\gamma_{5,6}}'\) to actor \(a_{env_1}\) and consumes and produces no tokens \((\text{cons}(a_{env_2}, a_{\gamma_{5,6}}'))(1) = \text{prod}(a_{\gamma_{5,6}}', a_{env_2})(1) = 0\) on the other channels connected to the composite actor.\(^{43}\) Consumption and produced rates of the two remaining CSDF phases are derived accordingly.

Additionally, in contrast to an SDF refinement, dependencies have to be added to the marked graph to force execution of the CSDF phases in sequentially ascending order, e.g., the dotted dashed arrows in Figure 4.19c from \(a_{\gamma_{5,6}}'\) to \(a_{\gamma_{5,6}}^2\), \(a_{\gamma_{5,6}}^2 \to a_{\gamma_{5,6}}^3\), and \(a_{\gamma_{5,6}}^3 \to a_{\gamma_{5,6}}'\) with an initial token. Hence, the condition for a safe CSDF cluster refinement has to be generalized from Theorem 4.1 as follows:

**Theorem 4.2 (Safe CSDF Cluster Refinement).** Given (1) a consistent and deadlock-free static subcluster \(g_\gamma\) of a DFG \(g\), (2) the marked graph expansion of the subcluster \(g_\gamma\), and (3) a grouping of actor firings consisting of an ordered set \(\{a_{\gamma_1}', a_{\gamma_2}', \ldots a_{\gamma_n}'\}\) of \(n\) coalescings of actor firings, then each coalescing of actor firings corresponds to one of the \(n\) CSDF phases of the generated composite actor. Furthermore, in order to force a sequentially ascending execution order for these \(n\) CSDF phases, the directed edges \(a_{\gamma_1}^1 \to a_{\gamma_2}^2, a_{\gamma_2}^2 \to a_{\gamma_3}^3, \ldots, a_{\gamma_{n-1}}^{n-1} \to a_{\gamma_n}'\)

\(^{43}\)In the following, a vector \(x = (x_1, x_2, \ldots x_n)\) is also considered to be a function from its index set \(I(x) = \{1, 2, \ldots n\}\) to its entries, that is \(x(i) = x_i\).
without any initial tokens as well as the edge \( a_1^\gamma \rightarrow a_1^\gamma \) with one initial token will be added to the marked graph expansion. Then, the refinement operation \( \xi_{CSDF}(g, g_\gamma) \) for the given grouping of actor firings is a safe cluster refinement operation if and only if the following three conditions hold:

- **Condition I:** The grouping of actor firings does not introduce any delay-less cycles in the modified marked graph expansion.

- **Condition II:** For each pair \((a_{\text{in}}^x, a_{\text{out}}^y) \in \mathcal{E}_\text{forward}\) of input/output actor firings, there exists a delay-less directed path from the input actor firing \( a_{\text{in}}^x \) to the output actor firing \( a_{\text{out}}^y \) in the original marked graph expansion of the subcluster \( g_\gamma \).

- **Condition III:** For each pair \((a_{\text{in}}^x, a_{\text{out}}^y) \in \mathcal{E}_\text{backward}\) of input/output actor firings, there exists a directed path with exactly one initial token from the input actor firing \( a_{\text{in}}^x \) to the output actor firing \( a_{\text{out}}^y \) in the original marked graph expansion of the subcluster \( g_\gamma \).

The set \( \mathcal{E}_\text{forward} \) is constrained such that it only contains pairs \((a_{\text{in}}^x, a_{\text{out}}^y)\) of input/output actor firings where

- \( a_{\text{in}} \in A_I \) is an input actor and \( a_{\text{out}} \in A_O \) is an output actor
- \( a_{\text{out}}^y \) is coalesced into some coalescing \( a_m^\gamma \in \{ a_1^\gamma, a_2^\gamma, \ldots, a_n^\gamma \}, 1 \leq m \leq n \)
- \( a_{\text{in}}^x \) is coalesced into some coalescing \( a_l^\gamma \in \{ a_1^\gamma, a_2^\gamma, \ldots, a_m^\gamma \}, 1 \leq l \leq m \) and \( a_{\text{in}}^x \neq a_{\text{out}}^y \)

The set \( \mathcal{E}_\text{backward} \) is constrained such that it only contains pairs \((a_{\text{in}}^x, a_{\text{out}}^y)\) of input/output actor firings where

- \( a_{\text{in}} \in A_I \) is an input actor and \( a_{\text{out}} \in A_O \) is an output actor
- \( a_{\text{out}}^y \) is coalesced into some coalescing \( a_m^\gamma \in \{ a_1^\gamma, a_2^\gamma, \ldots, a_n^\gamma \}, 1 \leq m \leq n \)
- \( a_{\text{in}}^x \) is coalesced into some coalescing \( a_{\gamma} \in \{ a_{\gamma}^m, a_{\gamma}^m+2, \ldots, a_{\gamma}^n \}, m < l \leq n \)

**Proof.** The key idea of Theorem 4.2 is that the CSDF refinement operation must not introduce any dependencies on tokens consumed by the composite actor for tokens produced by the composite actor that were not already present in the original cluster. The CSDF refinement operation introduces for all output actor firings two types of dependencies: (1) delay-less dependencies on all input actor firings in the same or sequentially previous \((1 \leq l \leq m)\) CSDF phases.

\[ \text{A delay-less directed path is a directed path where none of the edges of the path carries any initial tokens.} \]
4. Clustering

(cf. $\mathcal{E}_{\text{forward}}$), and (2) dependencies with exactly one initial token on all input actor firings in sequentially later ($m < l \leq n$) CSDF phases (cf. $\mathcal{E}_{\text{backward}}$). Hence, the second and the third condition in Theorem 4.2 ensures that equivalent dependencies are already present in the original marked graph expansion of the subcluster $g_7$. The first condition in Theorem 4.2 simply ensures that a valid schedule for all actor firings in the CSDF phases can be generated. 

To exemplify, the CSDF cluster refinement $(\tilde{g}_{\text{ex}4}, a'_6, \gamma'_r) = \xi_{\text{CSDF}}(g_{\text{ex}4}, g_{75,6})$ shown in Figure 4.19c is a safe cluster refinement. For the subcluster $g_{75,6}$, the set of input actors is $A_I = \{ a_6 \}$ and the set of output actors is $A_O = \{ a_5, a_6 \}$. Hence, for the grouping of actor firings as depicted in Figure 4.19c, delay-less dependencies are introduced for the pair of input/output actor firing $(a'_6, a'_5) \in \mathcal{E}_{\text{forward}} = \{ (a'_6, a'_5) \}$ and dependencies with exactly one initial token are introduced for the pairs of input/output actor firings $\mathcal{E}_{\text{backward}} = \{ (a'_6, a'_5), (a'_6, a'_5), (a'_6, a'_6) \}$. As can be seen by considering Figure 4.19a, for the pair $(a'_5, a'_6) \in \mathcal{E}_{\text{forward}}$ of input/output actor firing, the edge $a'_5 \rightarrow a'_6$ satisfies the existence of a delay-less path in the original marked graph expansion of the subcluster $g_7$. Furthermore, for all pairs of input/output actor firings $\mathcal{E}_{\text{backward}}$, there are equivalent edges in the original marked graph expansion of the subcluster $g_7$ carrying exactly one initial token.

However, the generated CSDF composite actor $a'_6$ has only one actor firing in each of its CSDF phases. This is a suboptimal situation from the perspective of overhead reduction. To optimize the reduction of the overall overhead, the length of the executed sequences of statically scheduled actor firings in each CSDF phase should be maximized. Fortunately, retiming can be employed to improve the situation. By firing actor $a_5$ once, the DFG depicted in Figure 4.18a is obtained. Instead of a refinement to an SDF composite actor (cf. Figures 4.18b and 4.21b), the CSDF composite actor $a''_{75,6}$ is now generated. To generate this CSDF composite actor, the grouping of the actor firings as depicted in Figure 4.21c is used. Hence, the CSDF composite actor $a''_{75,6}$ requires only the two CSDF phases as given below:

\[
\begin{align*}
\text{prod}(a'_{75,6}, a_{\text{env}1}) &= (0, 2) \\
\text{cons}(a_{\text{env}2}, a'_{75,6}) &= (1, 1) \\
\text{prod}(a'_{75,6}, a_{\text{env}2}) &= (1, 1)
\end{align*}
\]

The first CSDF phase is derived from the coalescing $a''_{75,6}$ which fires actor $a_6$ once, thus consuming one token ($\text{cons}(a_{\text{env}2}, a'_{75,6})(1) = 1$) on the channel from actor $a_{\text{env}2}$ to actor $a''_{75,6}$ as well as producing one token ($\text{prod}(a'_{75,6}, a_{\text{env}2})(1) = 1$) on the channel from actor $a''_{75,6}$ to actor $a_{\text{env}2}$. The second CSDF phase is derived from the coalescing $a''_{75,6}$ which fires actor $a_6$ once and, thus, leading
4.3 Cluster Refinement in Static Data Flow Graphs

Figure 4.21: Depicted is a marked graph expansion of the subcluster $g^r_{\gamma_{5,6}}$ as well as a coalescing and grouping of actor firings representing refinements of the subcluster to the SDF composite actor $a^r_{\gamma_{5,6}}$ and the CSDF composite actor $a^r'_{\gamma_{5,6}}$, respectively.

As previously, a composite actor for the original DFG $g_{\text{ex}4}$ including the retiming sequence can be generated. The resulting composite actor is depicted in Figure 4.22b. The transition $t_1$ implements the retiming operation while $t_2$ and $t_3$ correspond to the first and second CSDF phase, respectively. The composite actor $a^r_{\gamma_{5,6}}$ as well as the composite actor depicted in Figure 4.22b are generated by safe cluster refinement operations. Thus, these composite actors can be used to replace the cluster $g_{\gamma_{5,6}}$ ($g^r_{\gamma_{5,6}}$ for $a^r_{\gamma_{5,6}}$) for arbitrary cluster environments without introducing any artificial deadlocks.

However, in the general case, cluster refinement into a CSDF composite actor are unsafe. To exemplify, consider the cluster $g_{\gamma_{1,2,6}}$ in the two cluster environments $g_{\text{ex}5a}$ and $g_{\text{ex}5b}$, depicted in Figure 4.23. The problem stems from the fact that the actor $a_6$ has to fire first for the cluster environments $g_{\text{ex}5a}$ while the sequence $\langle a_1, a_2, a_2 \rangle$ has to fire first for cluster environments $g_{\text{ex}5b}$. Therefore, the decision which schedule sequence to choose depends on token availability on the input ports $i_1$ and $i_2$ of the cluster $g_{\gamma_{1,2,6}}$.

However, such flexibility cannot be represented via a CSDF composite actor. Furthermore, this situation cannot be salvaged via retiming as the four initial
4. Clustering

Figure 4.22: Composite actors derived by usage of retiming

(a) Composite actor $a'_{1,2,6}$

(b) Composite actor with included retiming for $g_{ex_4}$

Figure 4.23: Example of a cluster with no safe cluster refinement operation to a CSDF composite actor.

tokens, which are the reason for the flexibility to either fire $a_6$ or sequence $\langle a_1, a_2, a_2 \rangle$ first, in the cycle $a_1 \rightarrow a_6 \rightarrow a_1$ cannot be removed from the cluster $g_{1,2,6}$ via retiming. Instead, a QSS is required, which statically schedules the sequences $\langle a_1, a_2, a_2 \rangle$ and $\langle a_6 \rangle$ but postpones the decision which sequence to execute first to run time.

In the following, the situation of the cluster $g_{1,2,6}$ is considered in detail via its marked graph expansion depicted in Figure 4.24. Note that all coalescings are safe according to Theorem 4.1. Hence, splitting these coalescings will not enable the creation of a grouping of actor firings representing a safe cluster refinement operation to a CSDF actor according to Theorem 4.2 if this was not previously possible with the coalescings as given in Figure 4.24.

Thus, the remaining flexibility for building different CSDF composite actors is in the ordering of the CSDF phases $\langle a_1^1, a_2^2, a_2^2 \rangle$, $\langle a_6^1 \rangle$, $\langle a_6^2 \rangle$. In fact, the three CSDF composite actors $a'_{1,2,6}$, $a''_{1,2,6}$, and $a'''_{1,2,6}$ as given in Figures 4.25a to 4.25c represent the three possible valid orderings of the three CSDF phases $\langle a_1^1, a_2^2, a_2^2 \rangle$, $\langle a_6^1 \rangle$, $\langle a_6^2 \rangle$ to derive a CSDF composite actor for the cluster $g_{1,2,6}$.

In contrast to this, the grouping of actor firings as depicted in Figure 4.24a has an invalid ordering of the CSDF phases as indicated by the delay-less cycle.
Figure 4.24: Marked graph expansion of the cluster $g_{\gamma_{1,2,6}}$ and groupings of actor firings representing refinements of the cluster to different CSDF composite actors

(edges of the marked graph that are part of cycle are bolded) in violation of the first condition of Theorem 4.2. However, the groupings of actor firings depicted in Figures 4.24b to 4.24d are valid but do not correspond to a safe cluster refinement operation. This can be observed by noting that the groupings of actor firings for the three composite actors $a'_{\gamma_{1,2,6}}$, $a''_{\gamma_{1,2,6}}$, and $a'''_{\gamma_{1,2,6}}$ each have at least one violation of the second condition of Theorem 4.2.
4. Clustering

![Diagram of CSDF composite actors](image)

**Figure 4.25:** Depicted are various CSDF composite actors for the cluster $g_{7,1,2.6}$ as well as an FSM-based composite actor derived via a safe cluster refinement operation for the cluster $g_{7,1,2.6}$. As can be seen, the FSM-based QSS subsumes all three orderings of the CSDF phases $\langle a_1, a_2, a_3 \rangle$, $\langle a_6 \rangle$, and $\langle a_6 \rangle$ represented by the CSDF composite actors $a_{7,1,2.6}', a_{7,1,2.6}''$ and $a_{7,1,2.6}'''$.

To exemplify, for the CSDF composite actors $a_{7,1,2.6}'$ and $a_{7,1,2.6}''$, the ordering of CSDF phase $\langle a_6 \rangle$ before phase $\langle a_1, a_2, a_3 \rangle$ requires according to the second condition of Theorem 4.2 a delay-less directed path from the input actor firing $a_1^1$ to the output actor firing $a_3^1$ in the original marked graph expansion of the subcluster $g_{7,1,2.6}$. However, such a path is missing. Furthermore, for the CSDF composite actor $a_{7,1,2.6}'''$, the ordering of CSDF phase $\langle a_1, a_2, a_3 \rangle$ after phase $\langle a_1, a_2, a_3 \rangle$ according to the second condition of Theorem 4.2 a delay-less directed path from the input actor firing $a_1^1$ to the output actor firing $a_1^1$ in the original marked graph expansion of the subcluster $g_{7,1,2.6}$. But this path is also missing.

Hence, all possible CSDF composite actors with the three given phases are *unsafe*. And, thus, none of the three CSDF composite actors can be used in both cluster environments $g_{ex5a}$ and $g_{ex5b}$ without introducing at least in one of the cluster environments an artificial deadlock. For the cluster environments $g_{ex5a}$, the CSDF composite actor $a_{7,1,2.6}'''$ can be used for a deadlock-free execution.

118
However, for the cluster environments $g_{\text{cs}5b}$, all of the three CSDF composite actors $a'_{g_{71.2.6}}$, $a''_{g_{71.2.6}}$, or $a'''_{g_{71.2.6}}$ introduce an artificial deadlock. This seems to contradict the following theorem, which was also stated at the beginning of this subsection.

**Theorem 4.3** (CSDF Composite Actor Sufficiency For Known CSDF Cluster Environments). If a cluster environment $g_n$ containing only SDF or CSDF actors of a cluster $g_{\gamma}$ is known in advance, then a refinement into a CSDF composite actor $a_{\gamma}$ is sufficient to avoid the introduction of an artificial deadlock by the refinement operation $(g_n, a_{\gamma}) = \xi_{\text{CSDF}}(g_n, g_{\gamma})$ for the cluster environment $g_n$.

The contradiction can be resolved by considering CSDF composite actors containing more actor firings in all their CSDF phases than is strictly required by the repetition vector of the cluster from which the composite actor was derived. Such a CSDF composite actor can always be derived from an iteration of the CSDF graph $g_n$.

**Proof.** Given a consistent and deadlock-free SDF or CSDF graph $g_n$ and its subcluster $g_{\gamma}$, then a Periodic Static-Order Schedule (PSOS) $a^*$ for the graph $g_n$ can be calculated [BELP96, LM87b]. Furthermore, the sequence of actor firings $a$ of an iteration of the graph $g_n$ can be partitioned, that is $a = a_{g_n}^1 \land a_{g_n}^2 \land a_{g_n}^3 \land \ldots \land a_{g_n}^n \land a_{g_n}^n$, into (possibly empty) subsequences $a_{g_n}^j$ of actor firings containing only actor firings of the actors in the cluster environment and subsequences $a_{g_{\gamma}}^k$ of actor firings containing only actor firings of actors in the subcluster $g_{\gamma}$. Given such a partitioning, a CSDF composite actor with $n$ CSDF phases can be constructed by deriving each phase $a_{g_{\gamma}}^k$ from the subsequence $a_{g_{\gamma}}^k$. Then, a PSOS for the refined graph $g_{\bar{n}}$ is given by the following periodic sequence of actor firings: ($a_{g_n}^1 \land (a_{g_n}^1) \land a_{g_n}^2 \land (a_{g_n}^2) \land \ldots \land a_{g_n}^n \land (a_{g_n}^n) \land a_{g_n}^{n+1})^*$

To exemplify, for the SDF graph $g_{\text{cs}5b}$ shown in Figure 4.23b, the following PSOS $a^*$ can be derived via methods given in [LM87b] and its iteration partitioned into alternating subsequences containing only actor firings from the cluster environment and subsequences containing only actor firings from the cluster.

\[
\begin{align*}
a^* &= \langle a_1, a_2, a_3, a_4, a_5, a_6, a_2, a_1, a_2, a_3 \rangle^* \\
a &= a_{g_{\text{cs}5b}}^1 \land a_{g_{71.2.6}}^2 \land a_{g_{\text{cs}5b}}^3 \land a_{g_{71.2.6}}^4 \\
a_{g_{\text{cs}5b}}^1 &= \langle \rangle \\
a_{g_{71.2.6}}^2 &= \langle a_{\text{env}1}, a_{\text{env}2} \rangle \\
a_{g_{\text{cs}5b}}^3 &= \langle a_{\text{env}1}, a_{\text{env}2}, a_{\text{env}3} \rangle \\
a_{g_{71.2.6}}^4 &= \langle a_{\text{env}1}, a_{\text{env}2} \rangle
\end{align*}
\]
Hence, the CSDF composite actor for the cluster environment $g_{\text{ex}b}$ has the three CSDF phases $\langle a^1_1, a^2_2 \rangle$, $\langle a^1_1, a^2_2, a^3_6 \rangle$, and $\langle a^2_1, a^3_2, a^4_2 \rangle$.

Furthermore, due to the construction methodology employed in the proof, it will not introduce any deadlocks in the refined graph $\tilde{g}_{\text{ex}b}$. Note, however, that the resulting CSDF composite actor contains twice the number of actor firings as required by the repetition vector $r^\text{rep}_{\gamma_{1,2,6}} = (r^\text{rep}_1, r^\text{rep}_2, r^\text{rep}_6) = (1, 2, 2)$ of the cluster $g_{\gamma_{1,2,6}}$.

Note also that the CSDF composite actor for the cluster environment $g_{\text{ex}b}$ can be assembled from two cycles $c'$ and $c''$ as implemented in the composite actors $a'^{\gamma_{1,2,6}}$ and $a''^{\gamma_{1,2,6}}$ as given in Figure 4.25, respectively. Finally, one can observe that the three CSDF composite actors $a'^{\gamma_{1,2,6}}$, $a''^{\gamma_{1,2,6}}$, and $a'''^{\gamma_{1,2,6}}$ can be combined into the composite actor depicted in Figure 4.25d implementing an automata-based QSS supporting all possible combinations of the cycles in the CSDF composite actors $a'^{\gamma_{1,2,6}}$, $a''^{\gamma_{1,2,6}}$, and $a'''^{\gamma_{1,2,6}}$.

On the other hand, as demonstrated by the cluster $g_{\gamma_{1,2,6}}$ and the two cluster environments $g_{\text{ex}a}$ and $g_{\text{ex}b}$ shown in Figure 4.23, a refinement into a CSDF composite actor is insufficient to guarantee a safe cluster refinement operation for an arbitrary static cluster. That is to say, a refinement into a CSDF composite actor is insufficient to prevent deadlock for all possible cluster environments of the arbitrary static cluster even if the cluster environment contains only SDF or CSDF actors.

### 4.4 Cluster Refinement in Dynamic Data Flow Graphs

In this section, clustering of a DFG that consists of a mix of dynamic and static actors is considered. Such a DFG can be obtained via the classification methodology presented in Section 3.7. This methodology presents a sufficient condition to identify static data flow actors in a network graph. An example of such a classified network graph is given in Figure 4.26. It contains both static data flow actors (shaded vertices), i.e., HSDF, SDF, and CSDF actors with constant consumption and production rates, as well as dynamic data flow actors like the Parser and the ImageSink which are modeled by Kahn processes [Kah74].

The shaded actors, JPEG Source, IDCT, and Inverse ZigZag represent static actors (or subclusters containing only static actors, e.g., the IDCT subcluster from Figure 4.10) while the remaining actors could not be classified as static actors. In the following, safe cluster refinements according to Definition 4.3 are considered for static subclusters of a DFG consisting of a mix of dynamic and static actors. Each cluster refinement of a static subcluster results in a composite actor implementing a QSS for the actors in the subcluster.
As shown in Section 4.3, only safe cluster refinement operations can be considered for refining a static subcluster in an unknown cluster environment or a dynamic data flow cluster environment. Otherwise, the refinement operation may introduce artificial deadlocks into the resulting refined DFG.\textsuperscript{45}

Here, the main goal of the refinement operation is the improvement of the performance of the whole graph in terms of latency and throughput. This goal is achieved via the more efficient QSS scheme implemented by the composite actor in contrast to the overhead (cf. Section 4.1) induced by the dynamic scheduling of the static actors inside the cluster.

Of course, a safe cluster refinement to SDF or CSDF actors as presented in Sections 4.3.1 and 4.3.2 also reduces the overhead by implementing a PSOS for the contained actors. However, the design space for safe cluster refinements to SDF or CSDF actors is limited by the Theorems 4.1 and 4.2. In contrast to this, the design space for the safe cluster refinement operations presented in Sections 4.4.2 and 4.4.3 is significantly larger. Indeed, the only requirement for these safe cluster refinement operations is that the satisfaction of Definition 4.2 does not require an accumulation of an infinite amount of tokens in at least one channel contained in the refined cluster. This condition will be called the clustering condition in the following.

Nonetheless, the cluster refinement operations presented in Sections 4.4.2 and 4.4.3 cannot refine arbitrary static clusters. Hence, the problem of partitioning the set of all static actors in the network graph into a set of static clusters conforming to the clustering condition has to be solved first. To exemplify, the actors IDCT and Inverse ZigZag of the Motion-JPEG decoder network graph form a static cluster conforming to the clustering condition while the static JPEG Source actor alone is a degenerated static cluster containing only one actor.

\textsuperscript{45}The original DFG is assumed to be deadlock-free. This, however, cannot be detected due to the undecidability [Buc93] of deadlock freedom for dynamic DFGs.
However, in the general case, partitioning the set of all static actors into conforming static subclusters is a complex task of which the details are presented in the next section. After such a partitioning has been computed, the refinement operation presented in Sections 4.4.2 and 4.4.3 can be applied for each static cluster in the partitioning. The scheduling step presented in Section 4.4.4 translates the partial repetition vectors contained in the QSSs that are computed by the algorithms presented in Sections 4.4.2 and 4.4.3 into sequences of statically scheduled actor firings.

Definition 4.2 requires that the cluster is connected via channels of infinite size to its cluster environment. If this no longer holds, the channel sizes of the channels connecting the cluster and its environment have to be adjusted. This adjustment is presented in Section 4.4.6. Finally, after the model refinement presented in this chapter have been performed, the methods for code generation as presented in Chapter 5 can be applied.

4.4.1 Clustering Condition

The cluster refinement operations presented in Sections 4.4.2 and 4.4.3 are based on the generation of a finite state space representation of a QSS for a cluster. Hence, the number of states a cluster is required to exhibit in order to conform to Definition 4.2 must be finite. To ensure such a finite state space, the clustering condition given below must be satisfied for any cluster.

Definition 4.4 (Clustering Condition [FKH+08*]). For a cluster $g_{\gamma}$ and for for each pair $(a_{\text{in}}, a_{\text{out}})$ of input/output actors, there must exist a directed path $p$ from the input actor $a_{\text{in}}$ to the output actor $a_{\text{out}}$ in the cluster $g_{\gamma}$. In other words, $\forall (a_{\text{in}}, a_{\text{out}}) \in g_{\gamma}.A_I \times g_{\gamma}.A_O : \exists p = (a_1, a_2, \ldots, a_n) \in g_{\gamma}.A^* \text{ such that } a_1 = a_{\text{in}}, a_n = a_{\text{out}}, \text{ and } \forall k, 1 \leq k < n : \exists c \in g_{\gamma}.C : (a_k.O \times \{c\}) \cap g_{\gamma}.E \neq \emptyset \land (\{c\} \times a_{k+1}.I) \cap g_{\gamma}.E \neq \emptyset$.

To exemplify, the cluster $g_{\gamma_{5,6,7}}$ depicted in Figure 4.27 is considered. More formally, the cluster $g_{\gamma_{5,6,7}} = (P, V, E)$ is defined as follows:

$$\begin{align*}
P &= I \cup O = \{i_1\} \cup \{o_1, o_2\} \\
V &= A \cup C = \{a_5, a_6, a_7\} \cup \{c_{7 \rightarrow 6}, c_{6 \rightarrow 5}\} \\
E &= \{(i_1, a_6.i_1), (a_6.o_1, o_1), (a_5.o_1, o_2)\} \cup \\
&\quad \{(a_7.o_1, c_{7 \rightarrow 6}), (c_{7 \rightarrow 6}, a_6.i_2), (a_6.o_2, c_{6 \rightarrow 5}), (c_{6 \rightarrow 5}, a_5.i_1)\}
\end{align*}$$

Now, to check if the cluster $g_{\gamma_{5,6,7}}$ conforms to the clustering condition, its set of input actors $A_I = \{a_6\}$ and its set of outputs actors $A_O = \{a_5, a_6\}$.

The delay function has initially been introduced (cf. Definition 2.1 on page 8) as a function defining a sequence of initial tokens for each channel. Hence, to be technically correct, the length of the sequence has to be computed via the ' # '-operator.
4.4 Cluster Refinement in Dynamic Data Flow Graphs

![Figure 4.27](image)

Figure 4.27: Cluster \( g_{75.9.7} \) derived from adding an intermediate actor \( a_7 \) to the cluster \( g_{75.6} \) depicted in Figure 4.22 on page 116.

have to be determined. Then for each of the two pairs \( (a_6, a_6) \) and \( (a_6, a_5) \) of input/output actors, the existence of a directed path has to be verified. For the first pair \( (a_6, a_6) \), this is trivially true. For the second pair \( (a_6, a_5) \), the directed path from the input actor \( a_6 \) to the output actor \( a_5 \) is \( p = (a_6, a_5) \) via the channel \( c_{6 \rightarrow 5} \). Hence, the subcluster \( g_{75.6.7} \) does conform to the clustering condition.

In contrast to this, the cluster \( g_{78.9} \) depicted in Figure 4.28 does not conform to the clustering condition. More formally, the cluster \( g_{78.9} = (P, V, E) \) is defined as follows:

\[
\begin{align*}
P &= I \cup O = \{i_1, i_2\} \cup \{o_1\} \\
V &= A \cup C = \{a_8, a_9\} \cup \{c_{8 \rightarrow 9}\} \\
E &= \{(i_1, a_8, i_1), (a_8, o_1, o_1), (a_8, a_2, c_{8 \rightarrow 9}), (c_{8 \rightarrow 9}, a_9, i_1), (i_2, a_9, i_2)\}
\end{align*}
\]

The set of input and output actors of the cluster \( g_{78.9} \) can be determined as \( A_I = \{a_8, a_9\} \) and \( A_O = \{a_8\} \), respectively. Hence, the existence of a directed path for the two pairs \( (a_8, a_8) \) and \( (a_9, a_8) \) of input/output actors has to be verified. For the first pair \( (a_8, a_8) \), this is trivially true. However, for the second pair \( (a_9, a_8) \), no such directed path from the input actor \( a_9 \) to the output actor \( a_8 \) exists. Thus, the subcluster \( g_{78.9} \) does not conform to the clustering condition.

For a potential cluster \( g \) that does not satisfy the clustering condition, the set of static actors \( g.A \) can always be partitioned into subsets by consecutively removing channels with the unbounded token accumulation problem from the cluster \( g \). This operation will result in a decomposition of the potential cluster to a set of clusters conforming to the clustering condition.

To exemplify, the DFG \( g_{ex4} \) depicted in Figure 4.29 is considered. Consumption and production rates of the static actors as well as initial tokens are not...
4. Clustering

Figure 4.28: Example cluster $g_{7,9}$ for non-conformance to the clustering condition

shown in Figure 4.29 as this information is not required for the cluster partitioning heuristic presented in this section. Note also that the clusters depicted in Figures 4.11, 4.12, 4.17, 4.23, 4.27, and 4.28 are all subgraphs of $g_{ex6}$ and are all—apart from $g_{7,9}$—valid clusters conforming to the clustering condition.

Figure 4.29: Example DFG $g_{ex6}$ consisting of a mix of dynamic and static (shaded vertices) data flow actors

The goal of the presented heuristic is the formation of as large as possible clusters conforming to the clustering condition given in Definition 4.4. The heuristic starts by partitioning the set of static actors $g.A_S \subseteq g.A$ of a DFG $g$ into cluster candidates. This is done by removing all dynamic actors from the DFG and finding connected components in the resulting graph. These connected components form the cluster candidates. In the general case, however, these cluster candidates will not satisfy the clustering condition. Hence, they have to be par-
tioned into clusters satisfying the clustering condition by removing edges from the cluster candidates. Removal of edges will lead to smaller connected components forming smaller cluster candidates. The heuristic will terminate when all cluster candidates satisfy the clustering condition. Note that the heuristic will always terminate as the removal of all edges will result in the situation that each static actor is contained in its own unique cluster conforming to the clustering condition. This would even work if the edges to remove are chosen randomly. In the following, a greedy heuristic is presented that minimizes the number of removed edges:

- The heuristic starts by first replacing all dynamic actors in the original dynamic DFG by generic $a_{\text{src}}$ and $a_{\text{snk}}$ placeholders. To exemplify, the DFG $g_{\text{ex}}$ is considered. The resulting graph after placeholder substitution is depicted in Figure 4.30.

![Figure 4.30](image)

**Figure 4.30:** Modified graph $g_{\text{ex}}$ after replacement of all dynamic actors in with the $a_{\text{src}}$ and $a_{\text{snk}}$ placeholders

- The heuristic proceeds with the identification of cluster candidates. These will be determined by removing the $a_{\text{src}}$ and $a_{\text{snk}}$ placeholder actors and finding connected components in the resulting graph. These connected components form the cluster candidates. To exemplify, the graph depicted in Figure 4.30 is again considered. After removal of $a_{\text{src}}$ and $a_{\text{snk}}$, the graph
decays into the two connected components $g_{\gamma_{1,7,10-13}}$ and $g_{\gamma_{8,9}}$ as depicted in Figure 4.31.

**Figure 4.31:** Initial cluster candidates for $g_{ex_6}$

- After the cluster candidates have been identified, the set of input and output actors for each cluster candidate is computed. To exemplify, for the cluster candidates $g_{\gamma_{1,7,10-13}}$ and $g_{\gamma_{8,9}}$ depicted in Figure 4.31, the set of input and output actors is as follows:

  \[
  \begin{align*}
  g_{\gamma_{1,7,10-13}}.A_I &= \{a_1, a_4, a_{11}\} \\
  g_{\gamma_{1,7,10-13}}.A_O &= \{a_6, a_{11}, a_{12}\} \\
  g_{\gamma_{8,9}}.A_I &= \{a_8, a_9\} \\
  g_{\gamma_{8,9}}.A_O &= \{a_8\}
  \end{align*}
  \]

As can be seen, input actors are connected directly to the $a_{src}$ placeholder actor while output actors are connected directly to the $a_{sink}$ placeholder actor. Furthermore, the set of input and output actors of a cluster are not necessarily disjoint sets, e.g., the actors $a_8$ and $a_{11}$ are contained in both input and output actor sets of their respective clusters.

- After the sets of cluster input and output actors have been determined for each cluster candidate, each actor will be annotated with the a set
of cluster input actors $A_I$ and a set of cluster output actors $A_O$. To exemplify, the annotated graph depicted in Figure 4.32 is considered. The sets of cluster input and output actors are annotated as $A_I = \{ \ldots \}$ and $A_O = \{ \ldots \}$ at the corresponding actors of the cluster candidates, e.g., $A_I = \{ a_8 \}$ and $A_O = \{ a_8 \}$ at actor $a_8$ of the cluster candidate $g_{7,8,9}$. In the following, $a.A_I$ and $a.A_O$ will be used to refer to the two sets of cluster input and output actors which are annotated at actor $a$.

The annotated sets $a.A_I$ and $a.A_O$ correspond to the set of cluster input actors that are reachable by traversing the graph in backward direction starting from actor $a$ as well as the set of cluster output actors that are reachable by traversing the graph in forward direction starting from actor $a$. To exemplify, for actor $a_9$ no output actors are reachable by a forward traversal of the graph as the actor has no outgoing edges. However, the input actor $a_8$ is reachable by a backward traversal of the graph via the edge $a_8 \rightarrow a_9$. Hence, the annotated sets are $a_9.A_O = \emptyset$ and $a_9.A_I = \{ a_8 \}$. An actor is always reachable from itself. Thus, the cluster input and output actor $a_8$ is itself contained in both its annotated sets. However, the cluster input actor $a_9$ is not reachable by a backward traversal of the graph as no directed path from $a_9$ to $a_8$ is contained in the cluster candidate $g_{7,8,9}$. Hence, the annotated sets of actor $a_8$ are $a_8.A_O = \{ a_8 \}$ and $a_8.A_I = \{ a_8 \}$. 

![Figure 4.32: DFG $g_{8,6}$ with annotated predecessor cluster input actors $A_I$ and successor cluster output actors $A_O$](image)
4. Clustering

- After all actors of the cluster candidates are annotated with their sets of cluster input and output actors, equivalence classes are computed for each cluster candidate. An equivalence class $A_{EC_n}$ is a set of actors of a cluster candidate with identical annotated sets $A_I$ and $A_O$. More formally, the equivalence class $A_{EC_n}$ of the cluster candidate $g_\gamma$ containing actor $a'$ is

$$A_{EC_n} = \{ a \in g_\gamma, A \mid a.A_I = a'.A_I \land a.A_O = a'.A_O \}.$$

An equivalence class has the property that removal of edges connecting actors of the equivalence class will never be required in order to partition the static actors into clusters satisfying the clustering condition. Hence, these edges can be removed from consideration by coalescing all actors of an equivalence class into a single vertex.

To exemplify, Figure 4.33 depicting the equivalence classes derived from the annotated DFG from Figure 4.32 is considered. As can be seen in Figure 4.32, the actors $a_1, a_2,$ and $a_6$ have identical annotated sets

$$A_I = \{ a_1, a_4 \} \quad A_O = \{ a_6, a_{11}, a_{12} \}.$$

Hence, all three actors are represented by the single equivalence class $A_{EC3}$ in Figure 4.33.

$$A_{EC3} = \{ a_1, a_2, a_6 \} \quad A_I = \{ a_1, a_4 \} \quad A_O = \{ a_6, a_{11}, a_{12} \}$$

$$A_{EC1} = \{ a_3, a_4 \} \quad A_I = \{ a_4 \} \quad A_O = \{ a_6, a_{11}, a_{12} \}$$

$$A_{EC2} = \{ a_7 \} \quad A_I = \emptyset \quad A_O = \{ a_6, a_{11}, a_{12} \}$$

$$A_{EC4} = \{ a_5, a_{10} \} \quad A_I = \{ a_1, a_4 \} \quad A_O = \{ a_{11}, a_{12} \}$$

$$A_{EC5} = \{ a_{11}, a_{12}, a_{13} \} \quad A_I = \{ a_{11}, a_{11} \} \quad A_O = \{ a_{11}, a_{12} \}$$

$$A_{EC6} = \{ a_8 \} \quad A_I = \{ a_8 \} \quad A_O = \emptyset$$

$$A_{EC7} = \{ a_9 \} \quad A_I = \{ a_8, a_9 \} \quad A_O = \emptyset$$

**Figure 4.33:** Equivalence classes for the DFG $g_{ex6}$

- After the computation of the set of equivalence classes, clashes between different equivalence classes are determined. If two equivalence classes $A_{EC_n}$ and $A_{EC_m}$ have a clash, then all paths connecting $A_{EC_n}$ and $A_{EC_m}$ have to be cut. The clash relation is a symmetric relation that evaluates to true if the following three conditions are satisfied:

$$47$$ However, equivalence classes may change if edges are removed as this will change the set of cluster input and output actors of the updated cluster candidates as well as the annotated sets.
4.4 Cluster Refinement in Dynamic Data Flow Graphs

- $A_{ECn}$ and $A_{ECm}$ have a directed path w.l.o.g. from $A_{ECn}$ to $A_{ECm}$
- $A_{ECn}.A_I \neq A_{ECm}.A_I \land A_{ECn}.A_O \neq A_{ECm}.A_O$
- $\exists a_{out} \in A_{ECn}.A_O \setminus A_{ECm}.A_O : A_{ECm}.A_I \not\subseteq a_{out}.A_I$

In other words, two equivalence classes $A_{ECn}$ and $A_{ECm}$ do not clash if there is no directed path from $A_{ECn}$ to $A_{ECm}$ or from $A_{ECm}$ to $A_{ECn}$. In the following, it will be assumed that the directed path is from $A_{ECn}$ to $A_{ECm}$. Moreover, if the equivalence classes $A_{ECn}$ and $A_{ECm}$ have identical sets of predecessor input actors ($A_{ECn}.A_I = A_{ECm}.A_I$) or identical sets of successor output actors ($A_{ECn}.A_O = A_{ECm}.A_O$), then they do not clash. Finally, a clash is present if there exists a cluster output actor $a_{out} \in A_{ECn}.A_O \setminus A_{ECm}.A_O$ reachable from equivalence class $A_{ECn}$ but not from equivalence class $A_{ECm}$ that does not require a cluster input actor $a_{in} \in A_{ECm}.A_I \setminus a_{out}.A_I$ which is required by equivalence class $A_{ECm}$.

To exemplify, the cluster candidate $g_{\gamma8,9}$ and its two equivalence classes $A_{EC6}$ and $A_{EC7}$ are considered. This cluster has been used before (cf. Figure 4.28) as an example of non-conformance to the clustering condition. Hence, as expected, these two equivalence classes clash. The equivalence classes $A_{EC6}$ and $A_{EC7}$ are connected by a directed path from $A_{EC6}$ to $A_{EC7}$ and neither the set of cluster input actors nor the set of cluster output actors of $A_{EC6}$ and $A_{EC7}$ are identical. Furthermore, a cluster output actor $a_8$ reachable from $A_{EC6}$ but not from $A_{EC7}$ can be identified which does not require the cluster input actor $a_9$ which is required by equivalence class $A_{EC7}$. To exemplify, the clash relation clash for all equivalence classes of $g_{\text{ex}}$ is given in Figure 4.34.

- After the clashes between different equivalence classes have been determined, they must be resolved. Resolution of clashes is performed by cutting all paths between clashing equivalence classes. To remove the least number of edges possible, a min-cut algorithm [SW97] is used to cut these paths.

4.4.2 Automata-Based Clustering

Now that the static clusters of the DFG are identified, these clusters will be replaced by their corresponding composite actor implementing a QSS for the static actors contained in the cluster. In this section, an algorithm is presented

\[\text{Note that the graph induced by the equivalence classes, e.g., Figure 4.33 for the cluster candidate under evaluation, must be an acyclic graph since all actors of a strongly connected component in the DFG of the cluster candidate must be in the same equivalence class. Thus, the equivalence classes $A_{ECn}$ and $A_{ECm}$ can simply be swapped if the directed path is from $A_{ECm}$ to $A_{ECn}$.}\]
4. Clustering

<table>
<thead>
<tr>
<th>$A_{ECn}$</th>
<th>$A_{EC1}$</th>
<th>$A_{EC2}$</th>
<th>$A_{EC3}$</th>
<th>$A_{EC4}$</th>
<th>$A_{EC5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{EC1}$</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>$A_{EC2}$</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>$A_{EC3}$</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>$A_{EC4}$</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>$A_{EC5}$</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

(a) The clash relation for the equivalence classes of the cluster candidate $g_{7,10-13}$

<table>
<thead>
<tr>
<th>$A_{ECn}$</th>
<th>$A_{EC6}$</th>
<th>$A_{EC7}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{EC1}$</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>$A_{EC2}$</td>
<td>t</td>
<td>f</td>
</tr>
</tbody>
</table>

(b) The clash relation for the equivalence classes of the cluster candidate $g_{7,8,9}$

Figure 4.34: The clash relation for all equivalence classes of $g_{ex6}$

which computes for a static cluster conforming to the clustering condition a cluster FSM representing a QSS. The actions on the transitions of the cluster FSM are sequences of statically scheduled actor firings that represent the static part of a QSS schedule while the run-time selection of the transition to execute is representing the dynamic part of the QSS schedule.

As mentioned previously, the cluster refinement operations presented in this and the next section are based on a finite state space representation of the cluster. Thus, the state of a cluster and its state space are considered in more detail. The set of states a cluster is required to exhibit in order to conform to Definition 4.2 will be called its cluster state space. For the cluster FSM representation of the QSS, the cluster state space corresponds to the state space of the cluster FSM.

For the purpose of a refinement of a static cluster, the state of a cluster can be abstracted from the actor functionality state (cf. Definition 3.1 on page 44) of the actors contained in the cluster because the actor functionality state of a static actor does not influence its communication behavior. Moreover, due to the independence of the communication behavior of the static actors in the cluster from the actual values which are consumed and produced by these actors, the state of a cluster can be abstracted from the sequences of token values contained in the channels of the cluster to the number of tokens contained in the channels. Thus, the state of a cluster can be given by the number of tokens in all the channels contained in the cluster and the actor FSM state of each actor contained in the cluster. The state space of each actor FSM is by definition finite. Hence, the only possibility of an infinite cluster state space is a possible unbounded accumulation of tokens on a channel contained in the cluster.

However, an alternative and more concise way to specify the state of a cluster is used. More formally, the following definition of a cluster state can be given:

**Definition 4.5 (Cluster State [FZHT13*])**. The cluster state of a cluster $g_\gamma$ is encoded by a vector $q = (\eta_{in1}, \eta_{in2}, \ldots, \eta_{in|A_I|}, \eta_{out1}, \eta_{out2}, \ldots, \eta_{out|A_O|}) \in \mathbb{N}_{0}^{|A_I| \cup |A_O|}$
containing the number of accumulated actor firings $\eta_{in_1}, \eta_{in_2}, \ldots, \eta_{in_{|A_I|}}$ of all input actors $A_I = \{a_{in_1}, a_{in_2}, \ldots, a_{in_{|A_I|}}\}$ of the cluster and the number of accumulated actor firings $\eta_{out_1}, \eta_{out_2}, \ldots, \eta_{out_{|A_O|}}$ of all output actors of the cluster $\{a_{out_1}, a_{out_2}, \ldots, a_{out_{|A_O|}}\}$. Note that some of the actor firings $\eta_{in}$ and $\eta_{out}$ may overlap as the set of input actors and the set of output actors are in general not disjoint sets.

Hence, the initial cluster state $q_0$ of a cluster is encoded by the all zero vector $0$. Indeed, this representation of a cluster state does abstract from even more information compared to a representation based on the state of each actor FSM and the number of tokens in all channels contained in the cluster. The missing information is the number of actor firings of all the intermediate actors of the cluster. Intermediate actors of a cluster are those actors of the cluster that are neither input nor output actors.

Fortunately, this information is largely redundant for the cluster refinement operations in Sections 4.4.2 and 4.4.3 as only the interaction with the cluster environment through the cluster’s input and output actors can be used to adequately capture the state of the cluster. However, in Section 4.4.4, where the sequences of statically scheduled actor firings of the QSS are determined, the number of firings $\eta \in \mathbb{N}_0^{|A|}$ of all actors in the cluster $g_{\gamma}$ corresponding to a cluster state $q \in \mathbb{N}_0^{|A_I \cup A_O|}$ is required.

This information will be given by the function $\text{firings} : \mathbb{N}_0^{|A_I \cup A_O|} \rightarrow \mathbb{N}_0^{|A|}$. The value $\eta = \text{firings}(q)$ must, of course, contain the same number of firings for all input and output actors as contained in the cluster state $q$ from which it was derived, that is $\pi_{A_I \cup A_O}(\text{firings}(q)) = q$.\footnote{The notation $\pi_{\mathcal{I}}(\mathbf{x})$ will be used to denote a projection of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ to a subset of its entries given by an (ordered) subset $N \subseteq \mathcal{I}(\mathbf{x})$ of its index set $\mathcal{I}(\mathbf{x}) = \{1, 2, \ldots, n\}$. The order of the index in the subset $N$ determines the order in which the entries $x_k$ appear in the resulting projected vector $\pi_N(\mathbf{x})$, e.g., $\pi_{\{3, 5, 4\}}(x_1, x_2, x_3, x_4, x_5, x_6) = (x_3, x_5, x_4)$.}

To exemplify, the cluster $g_{7,6,7}$ depicted in Figure 4.27 is considered again. This cluster is derived from the cluster $g_{7,6,6}$ (cf. Figure 4.22 on page 116) by adding an intermediate actor $a_7$. Due to identical input and output actors, an analogous topology connecting them, as well as identical repetition counts $\eta_{\text{rep}}$ and $\eta_0$ in the repetition vectors $\eta_{\text{rep}} = (\eta_5^\text{rep}, \eta_6^\text{rep}, \eta_7^\text{rep}) = (1, 2, 1)$ and $\eta_0 = (\eta_5, \eta_6, \eta_7) = (1, 0, 0)$, the clusters $g_{7,6,6,7}$ and $g_{7,6,6}$ will have the same cluster state representation $q = (\eta_5, \eta_6, \eta_7)$ and the same three cluster states \{ $q_0 = (0, 0), q_1 = (1, 0), q_2 = (1, 1)$ \} (cf. Figure 4.22b on page 116).

In contrast to cluster $g_{7,6,6}$, which is lacking any intermediate actors, the cluster $g_{7,6,6,7}$ has multiple interpretations, that is multiple valid function values for $\text{firings}(q_1)$ and $\text{firings}(q_2)$. An example of this is depicted in Figure 4.35. As can be seen, the cycle $q_1 \rightarrow q_2 \rightarrow q_1$ must always contain one firing of the
intermediate actor \( a_7 \) as required by the repetition count \( \eta_7^{\text{rep}} = 1 \). However, the transition where this firing is include can be freely chosen.

\[
\#_1 \geq 1 \land \#_0 \geq 1 \land \#_2 \geq 2 / \#_1 \geq 1 \land \#_0 \geq 1 \land \#_2 \geq 2 / \#_1 \geq 1 \land \#_0 \geq 1 \land \#_2 \geq 2 /
\]

(a) firings \( (q_1) = (1,0,0) \); (b) firings \( (q_1) = (1,0,1) \); (c) firings \( (q_1) = (1,0,0) \); firings \( (q_2) = (1,1,0) \); firings \( (q_2) = (1,1,1) \); firings \( (q_2) = (1,1,1) \);

**Figure 4.35:** Shown are three different interpretations for the cluster states \( q_1 \) and \( q_2 \) as well as the *cluster FSMs* corresponding to these interpretations. Note that the cluster state \( q_0 \) will always be firings \( (q_0) = (\eta_5, \eta_6, \eta_7) = (0,0,0) \).

Furthermore, if the vector \( \eta_{\text{src} \rightarrow \text{dst}} \) is used to denote the number of actor firings executed by the transition \( q_{\text{src}} \rightarrow q_{\text{dst}} \), then below given equation holds:

\[
\text{firings}(q_{\text{src}}) + \eta_{\text{src} \rightarrow \text{dst}} \mod \eta_7^{\text{rep}} = \text{firings}(q_{\text{dst}}) \tag{4.3}
\]

To exemplify, the FSM depicted in Figure 4.35b is considered. For the transition \( q_0 \rightarrow q_1 \), the actors \( a_5 \) and \( a_7 \) are fired once. Hence, \( \eta_{0 \rightarrow 1} = (\eta_5, \eta_6, \eta_7) = (1,0,1) \) and substituting into Equation (4.3) results in the below given calculations:

\[
\text{firings}(q_0) + \eta_{0 \rightarrow 1} \mod \eta_7^{\text{rep}} = \text{firings}(q_1)
\]

\[
((0,0,0) + (1,0,1)) \mod (1,2,1) = \text{firings}(q_1)
\]

\[
(1,0,1) \mod (1,2,1) = \text{firings}(q_1)
\]

To demonstrate the wrap around in the cluster state space, as induced by the cyclic nature of SDF or CSDF graphs, the transition \( q_2 \rightarrow q_1 \) is considered. This transition fires each actor \( a_5, a_6, \) and \( a_7 \) exactly once, that is \( \eta_{2 \rightarrow 1} = (1,1,1) \).

---

\(^{50}\)The notation \( q \mod \eta \) is used to denote a modulo operation in a \( d \)-dimensional vector space. The modulo operation \( \mod \) is defined for vectors \( q \in \mathbb{Z}^d \) and \( \eta \in \mathbb{N}^d \). It computes the smallest vector \( q' = q \mod \eta \) such that \( q' \geq 0 \) and \( q \equiv q' \mod \eta \). More formally, \( q' \in \mathbb{N}^d \) such that \( \exists k \in \mathbb{Z} : q' = q - k \cdot \eta \land k \cdot \eta > k : q - k \cdot \eta \geq 0 \). Moreover, the \( \geq \)-relation between two vectors \( a \) and \( b \) with the same index set \( \mathcal{I}(a) = \mathcal{I}(b) \) denotes a partial order, i.e., \( a \geq b \iff \forall k \in \mathcal{I}(a) : a(k) \geq b(k) \).
Substituting the relevant vectors into Equation (4.3) results in the below given calculations:

\[(\text{firings}(q_2) + \eta_{2\to1}) \mod \eta_{\gamma,6} = \text{firings}(q_1)\]
\[((1,1,1) + (1,1,1)) \mod (1,2,1) = \text{firings}(q_1)\]
\[(2,2,2) \mod (1,2,1) = \text{firings}(q_i)\]
\[(1,0,1) = \text{firings}(q_i)\]

Without considering the cyclic nature of the cluster state space, the number of firings of all actors in the cluster for the destination state \(q_1\) would be calculated as \(\text{firings}(q_2) + \eta_{2\to1} = (1,1,1) + (1,1,1) = (2,2,2)\).

However, if each actor in the cluster \(g_\gamma\) has been fired exactly as many times as specified by its repetition vector \(\eta_{\gamma,rep}\), then the cluster must be in the same state as before. Hence, the cluster states after firing the actors in the cluster \(\eta + n \cdot \eta_{\gamma,rep} \forall n \in \mathbb{Z}\) times are all equivalent, e.g., \((2,2,2) \equiv (1,0,1) \mod \eta_{\gamma,rep}\).

To compute canonical representations for a number of actor firings \(\eta\) and a cluster state \(q\) the notations \(\eta \mod \eta_{\gamma,rep}\) and \(q \mod \pi_{A_I\cup A_O}(\eta_{\gamma,rep})\) are used, respectively.

**Cluster State Space**

In the following, the state space of a cluster \(Q_\gamma \subset \mathbb{N}_0^{\mid A_I \cup A_O\mid}\) will be defined precisely. The relevant point has been raised already in Section 4.2:

“The key idea for preserving deadlock freedom is to satisfy the worst case environment of the cluster. This environment contains for each output port \(o \in g_\gamma.O\) and each input port \(i \in g_\gamma.I\) a feedback loop where any produced token on the output port \(o\) is needed for the activation of an actor \(a \in g_\gamma.A\) connected to the input port \(i\).

In particular, postponing the production of an output token results in a deadlock of the entire system. Hence, to satisfy the equivalence condition from Definition 4.2, the composite actor produced by a cluster refinement must always produce the maximum number of output tokens possible from the consumption of a minimum number of input tokens.”

The above situation can be defined using the corresponding Kahn function \(s_{out} = \mathcal{K}_{g_\gamma}(s_{in})\) of the cluster \(g_\gamma\). The Kahn function by definition produces maximal sequences \(s_{out} = (s_{o_1}, s_{o_2}, \ldots s_{o_{\mid O\mid}})\) of output tokens on the cluster output ports \(\{o_1, o_2, \ldots o_{\mid O\mid}\} \in O\) from the given sequences \(s_{in} = (s_{i_1}, s_{i_2}, \ldots s_{i_{\mid I\mid}})\) of input tokens on the cluster input ports \(\{i_1, i_2, \ldots i_{\mid I\mid}\} \in I\). Hence, only
4. Clustering

the sequences \(s_{\text{in}}\) have to be minimized in order to define the tuple of signals \(s = (s_{\text{in}}, s_{\text{out}})\) where the maximum number of output tokens are produced from the consumption of a minimum number of input tokens, that is \(\{(s_{\text{in}}, K_{g\gamma}(s_{\text{in}})) \mid s_{\text{in}} \in S^{[l]} \wedge \exists s_{\text{in}} \subseteq s_{\text{in}} : K_{g\gamma}(s_{\text{in}}) = K_{g\gamma}(s_{\text{in}})\}\).

Finally, using the notation \(\text{state} : S^{[P]} \rightarrow Q\) to derive a cluster state \(q = \text{state}(s)\) from the tuple of signals \(s = (s_{\text{in}}, s_{\text{out}})\), the following definition of the cluster state space can be derived:

**Definition 4.6 (Cluster State Space).** Given a deadlock-free and consistent static subcluster \(g\gamma\) and its corresponding Kahn function \(K_{g\gamma}\), then its cluster state space is \(Q_{\gamma} = \left\{ \text{state}(s_{\text{in}}, K_{g\gamma}(s_{\text{in}})) \mod \pi_{A_{\text{rep}}(\eta_{\gamma})} \mid s_{\text{in}} \in S^{[l]} \wedge \exists s_{\text{in}} \subseteq s_{\text{in}} : K_{g\gamma}(s_{\text{in}}) = K_{g\gamma}(s_{\text{in}}) \right\} \cup \{q_0\} \subset \mathbb{N}^{[A_{\text{rep}}]}_0\). The initial cluster state \(q_0\) is then trivially the all zero vector \(0\) representing the fact that, in the beginning, no actor of the cluster has fired.

Each cluster state \(q \in Q_{\gamma}\) represents a state of the cluster that has consumed a minimum number of input tokens. Furthermore, each cluster state \(q \in Q_{\gamma} \setminus \{q_0\}\) has also produced a maximum number of output tokens from the consumption of a minimum number of input tokens. There may be one exception to the assertion that a cluster state always contains the maximal number of output actor firings, namely the initial cluster state \(0 = q_0 \in Q_{\gamma}\) (the all zero vector). This exception is due to the fact that excess initial tokens may be stored on the internal channels of the cluster in the initial state to permit the firing of cluster output actors without firing any cluster input actors. However, this exception can occur at most once per cluster as subsequent sequences of statically scheduled actor firings always produce the maximal number of output actor firings from a minimal number of input actor firings, and thus, tokens will not accumulate on internal channels. An example of such a situation are the three transitions \(q_0 \rightarrow q_1\) shown in Figure 4.35.

In general, the state transition graph of a cluster can be decomposed into (1) the initial state, (2) an acyclic part (except the initial state), and (3) a strongly connected component. The acyclic part represents possible retiming sequences to reach the cyclic behavior of the static DFG contained in the cluster. This behavior is represented by the strongly connected component. In detail, (1) the initial state may not have produced the maximum number of output tokens, whereas (2) each state of the acyclic part has produced the maximum number of output tokens possible, but may still contain some excess tokens on internal channels from the initial state, while (3) each state of the strongly connected component has both produced the maximum number of output tokens and all excess tokens from the initial state have been used up.
After having precisely defined the cluster state space, the connection between the cluster state space and the clustering condition can now be formally stated as follows:

**Theorem 4.4 (Finite Cluster State Space).** A connected, consistent, and deadlock-free static cluster $g_\gamma$ has a finite cluster state space if and only if the clustering condition given in Definition 4.4 holds for the cluster.

The proof of the above theorem is separated into two parts. First, it is proven that the clustering condition is a sufficient condition to guarantee a finite cluster state space.

**Proof.** Let $g_\gamma$ be a connected, consistent, and deadlock-free static cluster conforming to the clustering condition given in Definition 4.4. Thus, there exists a directed path in the cluster from each input actor of the cluster to each output actor of the cluster. Hence, for each output actor $a_n \in A_O$, there exists a firing count $\eta_{\text{bound},n}$ of this actor which requires all input actors $A_I$ to perform at least as many actor firings as encoded in the repetition vector $\eta_\gamma^{\text{rep}}$ of the cluster. That is $q(a_n) \geq \eta_{\text{bound},n} \implies \pi_{A_I}(q) \geq \pi_{A_I}(\eta_\gamma^{\text{rep}})$.\(^{51}\) Moreover, due to the requirement to produce a maximum number of output tokens possible from the consumption of a minimum number of input tokens, if all input actors of the cluster have fired at least as many actor firings as encoded in the repetition vector $\eta_\gamma^{\text{rep}}$ of the cluster, i.e., $\pi_{A_I}(q) \geq \pi_{A_I}(\eta_\gamma^{\text{rep}})$, then all output actors of the cluster will have done the same, i.e., $q \geq \pi_{A_I \cup A_O}(\eta_\gamma^{\text{rep}})$.

Hence, for all cluster states $q \in Q_\gamma$ and for all output actors $a_n \in A_O$, the number of output actor firings $q(a_n)$ is bounded by $\eta_{\text{bound},n}$. Otherwise, the following contradiction will result:

$$\exists a_n \in A_O : q'(a_n) \geq \eta_{\text{bound},n} \implies \pi_{A_I}(q') \geq \pi_{A_I}(\eta_\gamma^{\text{rep}}) \implies q' \geq \pi_{A_I \cup A_O}(\eta_\gamma^{\text{rep}}) \implies q' \neq q' \mod \pi_{A_I \cup A_O}(\eta_\gamma^{\text{rep}})$$

The existence of a bound for the number of input actor firings in a cluster state can immediately be derived from the existence of the bound on the number of output actor firings in the cluster state by defining the bound on the input actor firings for each input actor to be the minimal number of input actor firings required to execute $\eta_{\text{bound},n}$ output actor firings for each output actor. Hence, the number of input and output actor firings for an arbitrary cluster state $q \in Q_\gamma$ are all bounded. Thus, the cluster state space $Q_\gamma$ must be finite. \(\square\)

\(^{51}\)Please remember, a vector, e.g., $q = (\eta_{a_1}, \eta_{a_2}, \ldots, \eta_{a_n}, \ldots, \eta_{{a_I \cup A_O}})$, is always also considered to be a function from its index set, e.g., here $\mathcal{I}(q) = A_I \cup A_O$, to its entries, e.g., $q(a_n) = \eta_{a_n}$. Moreover, the $\geq$-relation between two vectors $a$ and $b$ with the same index set $\mathcal{I}(a) = \mathcal{I}(b)$ denotes a partial order, i.e., $a \geq b \iff \forall k \in \mathcal{I}(a) : a(k) \geq b(k)$. 135
4. Clustering

To complete the proof of Theorem 4.4, it is now shown that the clustering condition is a necessary condition to guarantee a finite cluster state space.

Proof. Let $g_\gamma$ be a connected, consistent, and deadlock-free static cluster not conforming to the clustering condition given in Definition 4.4. Due to the violation of the clustering condition, there exists at least one pair $(a_{in}, a_{out})$ of input/output actors without a directed path from the input actor $a_{in}$ to the output actor $a_{out}$ in the cluster $g_\gamma$. Hence, firings of the output actor $a_{out}$ are independent of firings of the input actor $a_{in}$. Thus, the state set of the cluster described by Definition 4.6 will contain an infinite amount of cluster states $q \in Q_\gamma$ where $q(a_{out}) \geq n, n \in \mathbb{N}$ but $q(a_{in}) = 0$. To exemplify, without loss of generality, let the cluster input port $i_1$ be connected to this input actor $a_{in}$ and no tokens are ever provided on this input port, but sufficient tokens are provided on all other input ports $i_2, \ldots, i_{|I|}$ for the output actor $a_{out}$ to fire $n$ times. Furthermore, due to the prerequisite that $g_\gamma$ is a connected graph, the two actors $a_{in}$ and $a_{out}$ are connected via a path $p$. Hence, at least one of the channels contained in the path $p$ connecting the actors $a_{in}$ and $a_{out}$ will accumulate an infinite number of tokens as $n$ approaches infinity.

Subsequently, the automata-based clustering algorithm that is presented in [FKH+08] will be reviewed. This algorithm computes the cluster FSM $a_\gamma \mathcal{R}$ (cf. Definition 4.1 on page 90) of a composite actor $a_\gamma \in \tilde{g}.A$ that is derived via the cluster refinement operator $\xi_{FSM}$, i.e., $\xi_{FSM}(g, g_\gamma) = (\tilde{g}, a_\gamma)$. This cluster FSM implements an automata-based representation of a QSS for the actors $g_\gamma.A$ contained in the static subcluster $g_\gamma$ of the original DFG $g$.

The automata-based clustering algorithm works in three steps: (Step 1) computes the set of input/output dependency function values by considering each output actor in isolation to obtain the minimal number of input actor firings required for $n$ firings of this output actor, (Step 2) derives the cluster state space $\mathcal{R}.Q \equiv Q_\gamma$ (cf. Definition 4.6) of the cluster FSM $a_\gamma.\mathcal{R}$ from the set of input/output dependency function values, and (Step 3) obtains the set of transitions $\mathcal{R}.T$ of the cluster FSM $a_\gamma.\mathcal{R}$ from the cluster state space $Q_\gamma$.

In order to avoid deadlocks, the worst case is assumed, i.e., each produced output token causes the activation of an actor in the subcluster via a feedback loop. Hence, it is required that the resulting QSS always produces a maximum number of output tokens from a minimum number of input tokens. The state of the cluster $g_\gamma$ after such a production is given by a cluster state $q \in Q_\gamma$. As an exhaustive evaluation (cf. Definition 4.6) is prohibitive, the cluster state space will be derived one output actor at a time by computing (Step 1) a set of input/output dependency function values for each output actor in isolation (cf. Definition 4.8). Afterwards, in (Step 2), the cluster state space is generated by a fixed-point operation (cf. Definition 4.10) that computes the pointwise
maximum for all pairs of input/output dependency function values until no more new values are generated. This fixed-point operation generates the necessary interleavings of output actor firings that have been previously neglected by only considering output actor firings in isolation. A cluster state is derived from an input/output dependency function value by determining (cf. Definition 4.9) the maximal number of output actor firings possible with the minimal number of input actor firings given by an input/output dependency function value. Finally, in (Step 3), the transition set $R.T$ of the cluster FSM is derived by partially ordering the cluster state set using the $>$-relation between two cluster states. A transition $q_{src} \rightarrow q_{dst}$ exists between two cluster states $q_{dst} > q_{src}$ if there is no further state $q'$ between, i.e., $\not\exists q' : q_{dst} > q' > q_{src}$, these two states. More precisely, the cluster FSM for a consistent and deadlock-free subcluster $g_\gamma$ is derived by following the three steps [FKH+08*] given below:

**Step 1: Computation of Input/Output Dependency Function Values**

First, the repetition vector $\eta^\text{rep}_\gamma$ for the subcluster $g_\gamma$ is computed in order to provide a termination criterion (cf. Equation (4.4) on page 139) for the computation of the set of input/output dependency function values in Definition 4.8. To exemplify, for the subcluster $g_{1,2,3}$ depicted in Figure 4.36 from [FZHT13*], the repetition vector is $\eta^\text{rep}_{1,2,3} = (\eta^\text{rep}_1, \eta^\text{rep}_2, \eta^\text{rep}_3) = (1, 1, 1)$, i.e., fire all three actors $a_1, a_2, \text{and } a_3$ once.

![Figure 4.36: Example network graph $g_{ex7}$ used for illustration of the automata-based clustering algorithm](image)

As token productions may not be postponed, the proposed algorithm uses the so-called input/output dependency function, which encodes the minimal number of input actor firings required for a given output actor $a_{out}$ and number of output actor firings $n$. With this information, it is known how many tokens are required on the input channels in order to enable the number of requested.
output actor firings. More formally, the input/output dependency function is defined as follows:

**Definition 4.7** (Input/Output Dependency Function [FKH +08*]). Given a static cluster $g_\gamma$, its input/output dependency function $\text{dep}: A_O \times \mathbb{N}_0 \rightarrow \mathbb{N}^{|A_I|}_0$ is a vector-valued function that associates with each output actor $a_{\text{out}} \in A_O$ and number $n \in \mathbb{N}_0$ of requested output actor firings, the minimal number of input actor firings $(\eta_{i_1}, \eta_{i_2}, \ldots \eta_{i_{|A_I|}}) \in \mathbb{N}^{|A_I|}_0$ required to produce the requested number $n$ of firings of the output actor $a_{\text{out}}$.

To exemplify, Table 4.1 lists values for the input/output dependency function $\text{dep}(a_{\text{out}}, n)$ of the static subcluster $g_{\gamma_{1,2,3}}$ depicted in Figure 4.36. In this case, the sets of input and output actors are $A_I = \{ a_1, a_3 \}$ and $A_O = \{ a_1, a_2, a_3 \}$, respectively. As can be seen from column $a_{\text{out}} = a_2$, no input actor firings are required ($(0, 0) = \text{dep}(a_2, 1)$) in order to fire the output actor $a_2$ once. However, for the second firing of the output actor $a_2$, both input actors $a_1$ and $a_3$ have to be fired once, i.e., $(1, 1) = \text{dep}(a_2, 2)$. The columns $a_{\text{out}} = a_1$ and $a_{\text{out}} = a_3$ represent similar information for the output actors $a_1$ and $a_3$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$a_{\text{out}} = a_1$</th>
<th>$a_{\text{out}} = a_2$</th>
<th>$a_{\text{out}} = a_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 0)</td>
<td>(0, 0)</td>
<td>(0, 1)</td>
</tr>
<tr>
<td>2</td>
<td>(2, 0)</td>
<td>(1, 1)</td>
<td>(0, 2)</td>
</tr>
<tr>
<td>3</td>
<td>(3, 1)</td>
<td></td>
<td>(1, 3)</td>
</tr>
</tbody>
</table>

**Table 4.1**: Input/output dependency function values $(\eta_1, \eta_3) = \text{dep}(a_{\text{out}}, n)$ for subcluster $g_{\gamma_{1,2,3}}$ from Figure 4.36.

Note that the number of input/output dependency function values that need to be considered is bounded by the repetition vector $\eta_{\gamma}^{\text{rep}}$ (cf. Equation (4.4)). In principle, only values $\text{dep}(a_{\text{out}}, n) \geq \pi_{A_I}(\eta_{\gamma}^{\text{rep}})$ need to be considered since values $\text{dep}(a_{\text{out}}, n) \geq \pi_{A_I}(\eta_{\gamma}^{\text{rep}})$ are redundant with respect to $(\text{mod } \eta_{\gamma}^{\text{rep}})$. However, this would complicate the computation of the set of transitions $\mathcal{R}T$ of the cluster FSM in (Step 3). Hence, for each output actor, the minimal input/output dependency function value that is greater than the repetition vector

---

52In [FKH+08*], the input/output dependency function as well as the cluster state space are defined in terms of minimum number of consumed tokens and maximum number of produced tokens on the cluster input and output ports. However, in order to have a consistent definition of cluster states and cluster state space for both the automata-based and the rule-based clustering algorithms, the automata-based clustering will be presented here using a cluster state space defined in terms of cluster input and output actor firings as used for the rule-based representation published in [FZHT11*]. Both representations are equivalent and can be transformed into each other by using the consumption and production rates of the input and output actors.
for the cluster input actors, e.g., the input/output dependency function values 
(3, 1), (1, 1), and (1, 3) in Table 4.1, will also be included. More formally, the set 
of input/output dependency function values for each output actor of a cluster 
is defined as follows:

**Definition 4.8 (Set of Input/Output Dependency Function Values).** Given a 
static cluster $g_\gamma$, its repetition vector $\eta^\text{rep}_\gamma$, and its input/output dependency function $\text{dep}$. Then, the set of input/output dependency function values $H_{\text{dep}}(a_{\text{out}})$ for a cluster output actor $a_{\text{out}}$ is given by:

$$H_{\text{dep}}(a_{\text{out}}) = \{ \text{dep}(a_{\text{out}}, n) \mid n \in \{1, 2, \ldots\}, \min\{n' \in \mathbb{N}_0 \mid \text{dep}(a_{\text{out}}, n') \geq \pi_{A_I}(\eta^\text{rep}_\gamma)\} \}$$  \hspace{1cm} (4.4)

Considering the DFG shown in Figure 4.36, these sets of input/output dependency function values for subcluster $g_{\gamma1,2,3}$ are: $\text{dep}(a_1) = \{(1, 0), (2, 0), (3, 1)\}$ for output actor $a_1$, $\text{dep}(a_2) = \{(0, 0), (1, 1)\}$ for output actor $a_2$, and $\text{dep}(a_3) = \{(0, 1), (0, 2), (1, 3)\}$ for output actor $a_3$.

**Step 2: Derivation of the Cluster FSM State Set**

Next, the cluster state space $Q_\gamma$ is derived from the set of input/output dependency function values. Each cluster state $q \in Q_\gamma$ encodes the minimal number of input actor firings $\pi_{A_I}(q)$ required for the output actor firings $\pi_{A_O}(q)$ given by the cluster state. By definition, an input/output dependency function value $\eta$ is already a minimal vector of input actor firings (cf. Equation (4.5)).

However, each cluster state $q \in Q_\gamma \setminus \{q_0\}$ also encodes the maximal number of output actor firings $\pi_{A_O}(q)$ that are possible with the given vector $\pi_{A_I}(q) = \eta$ of input actor firings. Therefore, in order to derive a cluster state $q$ from a vector $\eta$ of input actor firings, the maximal number of output actor firings (cf. Equation (4.6)) have to be determined for all output actors (cf. Equation (4.7)). This can be done efficiently, as the vector values of $\text{dep}$ for a given output actor $a_{\text{out}}$ are totally ordered under $\leq$, i.e., $\text{dep}(a_{\text{out}}, n) \leq \text{dep}(a_{\text{out}}, n + 1)$. More formally, a cluster state is derived from an input/output dependency function value as follows:

\[\text{The notation } \min X \{ \ldots \} \text{ is used to select the minimal value of a set } \{ \ldots \}.\]

\[\text{Remember that there may be one exception to the assertion that a cluster state always contains the maximal number of output actor firings, namely the initial cluster state } 0 = q_0 \in Q_\gamma \text{ (the all zero vector). This state is explicitly added (cf. Definition 4.6 on page 134) to the cluster state space. This exception is due to the fact that sufficient initial tokens may be stored on the internal channels of the cluster, e.g., channels } c_{1 \to 2} \text{ and } c_{3 \to 2} \text{ in the DFG } g_{\text{ex}} \text{ depicted in Figure 4.36, to permit the firing of cluster output actors without firing any cluster input actors, e.g., transition } q_0 \rightarrow q_1 \text{ of the composite actor depicted in Figure 4.38.}\]
4. Clustering

Definition 4.9 (Output Maximization Function \[\text{FKH}^+08^*\]). Given a static cluster \(g_\gamma\) and its input/output dependency function \(\text{dep}\), then the output maximization function \(\text{state}_{\text{dep}} : \mathbb{N}^{|A_I|} \rightarrow \mathbb{N}^{|A_I|}\) is a vector-valued function that associates with a vector \(\eta\) of input actor firings a cluster state \(q\) containing the maximal number of output actor firings \(\pi_{A_O}(q)\) enabled by the given vector \(\eta\) of input actor firings. This can be defined precisely as follows:

\[
\text{state}_{\text{dep}}(\eta) = q \text{ where } \pi_{A_I}(q) = \eta \quad (4.5)
\]

\[
\pi_{\{a_{\text{out}}\}}(q) = \max\{ n \in \mathbb{N}_0 | \eta \geq \text{dep}(a_{\text{out}}, n) \} \quad (4.6)
\]

\[
\forall a_{\text{out}} \in A_O \quad (4.7)
\]

To exemplify, in case of subcluster \(g_{71,22}\) shown in Figure 4.36, the set of initial input/output dependency states \(Q_{\text{ini}} = \{ \text{state}_{\text{dep}}(\eta) \mid \eta \in H_{\text{dep}}(a_{\text{out}}), a_{\text{out}} \in A_O \} \cup \{ q_0 \}\) is listed in the right half of Table 4.2. This set is directly derived from the set of input/output dependency function values shown in the left half of Table 4.2. As can be seen from column \(a_{\text{out}} = a_2\) on the right half of Table 4.2, no input actor firings are required \(((0,0) = \pi_{\{a_1, a_3\}}(0,1,0))\) in order to fire the output actor \(a_2\) once \((1 = \pi_{\{a_2\}}(0,1,0))\). However, for the second firing of the output actor \(a_2\), both input actors \(a_1\) and \(a_3\) have to be fired once, i.e., \((1,2,1) \in Q_{\text{ini}}\).

<table>
<thead>
<tr>
<th>(n)</th>
<th>(a_{\text{out}} = a_1)</th>
<th>(a_{\text{out}} = a_2)</th>
<th>(a_{\text{out}} = a_3)</th>
<th>(a_{\text{out}} = a_1)</th>
<th>(a_{\text{out}} = a_2)</th>
<th>(a_{\text{out}} = a_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,0)</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(1,1,0)</td>
<td>(0,1,0)</td>
<td>(0,1,1)</td>
</tr>
<tr>
<td>2</td>
<td>(2,0)</td>
<td>(1,1)</td>
<td>(0,2)</td>
<td>(2,1,0)</td>
<td>(1,2,1)</td>
<td>(0,1,2)</td>
</tr>
<tr>
<td>3</td>
<td>(3,1)</td>
<td>(1,3)</td>
<td>(1,3)</td>
<td>(3,2,1)</td>
<td>(1,2,3)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Input/output dependency function values \((\eta_1, \eta_3) = \text{dep}(a_{\text{out}}, n)\) from Table 4.1 and their corresponding cluster states \((\eta_1, \eta_2, \eta_3) = \text{state}_{\text{dep}}(\text{dep}(a_{\text{out}}, n)) \in Q_{\text{ini}}\)

However, as the calculation of the set of input/output dependency function values considers each output actor in isolation, the derived set \(Q_{\text{ini}}\) does not contain cluster states resulting from different interleavings of output actor firings. In detail, computing a pointwise maximum \(q_{\text{new}} = \max(q_n, q_m)\) of cluster states \(q_n, q_m \in Q_{\text{ini}}\) derived for different output actors \(a_n \neq a_m\) and \(a_n, a_m \in A_O\) might introduce a new cluster state \(q_{\text{new}} \notin Q_{\text{ini}}\) representing a state of the cluster that is reachable by an interleaving of the sequence of actor firings used to

---

55 This function is not explicitly defined in \[\text{FKH}^+08^*\], but it is present implicitly as part of the definition of the input/output state set in \[\text{FKH}^+08^*\].

56 The notation \(\max X\) is used to select the maximal value \(x \in X\) of a set \(X\).
reach the original cluster states \(q_n\) and \(q_m\). These interleavings are captured by the interleaved cluster state space \(Q_{lfp}\), which is defined as follows:

**Definition 4.10** (Interleaved Cluster State Space). The interleaved cluster state space \(Q_{lfp}\) is derived by (1) computing the least fixed point (cf. Equation (4.9)) of the interleave function and (2) converting each value of the resulting set representing the least fixed point into a cluster state by determining the maximal number of output actor firings (cf. Equation (4.8)) for this value. As always, the initial state \(q_0\) is an exception that has to be explicitly added (cf. Equation (4.10)). The interleave function is a monotonically increasing function that enlarges (cf. Equation (4.11)) the set \(H\) starting from the set of all input/output dependency function values (cf. Equation (4.12)) by computing the pointwise maximum (cf. Equations (4.13) to (4.14)) for each pair of input/output dependency function values. This can be defined precisely as follows:

\[
Q_{lfp} = \{ \text{state}_{dep}(\eta) \mid \eta \in \text{lfp}(H = \text{interleave}(H)) \} \cup \{ q_0 \} \tag{4.8}
\]

interleave\((H) = H \cup \{ \eta \in H_{dep}(a_{\text{out}}) \mid a_{\text{out}} \in A_O \} \cup \{ \max(\eta_1, \eta_2) \mid \eta_1, \eta_2 \in H \} \cup \{ \max(\eta_1, \eta_2) \mod \pi_{A_I}(\eta_{rep}^{\gamma}) \mid \eta_1, \eta_2 \in H \} \tag{4.9}
\]

For the subcluster \(g_{\gamma_1,\gamma_2,\gamma_3}\), the generated states are all greater than or equal to the repetition vector, i.e., \(\forall q \in \{ q_9, q_{10}, \ldots, q_{14} \} : q \geq \pi_{A_I \cup A_O}(\eta_{rep}^{\gamma_1,\gamma_2,\gamma_3})\), however, as demonstrated in [FKH+08*], cluster states \(\not\geq \pi_{A_I \cup A_O}(\eta_{rep}^{\gamma})\) can also be generated.

To exemplify Definition 4.10, consider Figure 4.37 that depicts Hasse diagrams for the cluster state sets \(Q_{ini}\) and \(Q_{lfp}\). As the interleave function is a monotonically increasing function, its least fixed point can be calculated iteratively applying the function to its result until the fixed point is reached, e.g., in case of subcluster \(g_{\gamma_1,\gamma_2,\gamma_3}\) shown in Figure 4.36, \(\text{interleave}(\text{interleave}(\text{interleave}(\emptyset)))\) computes the same set as \(\text{interleave}(\text{interleave}(\text{interleave}(\emptyset)))\). The resulting states \(Q_{ini} = \{ \text{state}_{dep}(\eta) \mid \eta \in \text{interleave}(\text{interleave}(\text{interleave}(\emptyset))) \} \) for one iteration of the interleave function are depicted in Figure 4.37a, while the final interleaved cluster states \(Q_{lfp} = \{ \text{state}_{dep}(\eta) \mid \eta \in \text{interleave}(\text{interleave}(\text{interleave}(\emptyset))) \} \) are shown in Figure 4.37b.

Comparing \(Q_{ini}\) and \(Q_{lfp}\), it can be seen that the interleaving adds the states \(q_9, q_{10}, \ldots, q_{14}\) to \(Q_{lfp}\) by pointwise maximum operations and computation of the maximal number of output actor firings for a cluster state, e.g., the state \(q_9 \in Q_{lfp}\) is derived from the states \(q_7\) and \(q_8\) from \(Q_{ini}\). The cluster state space can now be derived from the interleaved cluster state space as follows:

57The notation \(\text{lfp}(x = f(x))\) is used to select the minimal solution \(x\) of the equation \(x = f(x)\).

58For the subcluster \(g_{\gamma_1,\gamma_2,\gamma_3}\), the generated states are all greater than or equal to the repetition vector, i.e., \(\forall q \in \{ q_9, q_{10}, \ldots, q_{14} \} : q \geq \pi_{A_I \cup A_O}(\eta_{rep}^{\gamma_1,\gamma_2,\gamma_3})\), however, as demonstrated in [FKH+08*], cluster states \(\not\geq \pi_{A_I \cup A_O}(\eta_{rep}^{\gamma})\) can also be generated.
4. Clustering

Figure 4.37: Hasse diagrams for various cluster state sets of the subcluster $g_{\gamma_{1,2,3}}$ shown in Figure 4.36

Definition 4.11 (Cluster FSM State Set [FKH+08*]59). Given the interleaved cluster state space $Q_{\text{ilp}}$, the cluster FSM state set $\mathcal{R}.Q$ is derived by only considering cluster states not greater or equal to the repetition vector of the cluster, i.e., $\mathcal{R}.Q = \{ q \in Q_{\text{ilp}} \mid q \not\geq \pi_{A_1 \cup A_2}(\eta_{\gamma}) \}$. The initial state of the cluster FSM is the initial cluster state $\mathcal{R}.q_0 = q_0 = 0$ (the all zero vector).

In Figure 4.37, the cluster FSM state set $\mathcal{R}.Q$ has been marked by dashed groupings. These states $q_0, q_1, \ldots, q_5$ are also present in the cluster FSM of the composite actor $a_{\gamma_{1,2,3}}$ depicted in Figure 4.38.

Figure 4.38: The composite actor $a_{\gamma_{1,2,3}}$ replacing subcluster $g_{\gamma_{1,2,3}}$ depicted in Figure 4.36

59 There is no explicit definition in [FKH+08*], but it is contained in [FKH+08*] Step 3.2 where the cluster FSM state space is determined.
4.4 Cluster Refinement in Dynamic Data Flow Graphs

However, the question arises as to whether the cluster state sets given by Definitions 4.6 and 4.11 are equivalent. The following considerations can be used to answer this question.

Proof. The question of equivalence can be reduced to $Q_\gamma \subseteq R.Q$ and $R.Q \subseteq Q_\gamma$. Fortunately, $Q_\gamma \subseteq R.Q$ is a corollary to the semantic equivalence as given by Definition 4.2 and proven in [FZHT13*]. A replication of this proof is presented in Theorem 4.5 on page 167. In the proof of Theorem 4.5, it is shown that each cluster state $q_{ok} \in Q_\gamma$ is reachable from all cluster FSM states $q_{def} \in R.Q$ where $q_{ok} \geq q_{def}$. What is not proven is the absence of states $q_{def} \in R.Q \setminus Q_\gamma$. However, each state $q_{def} \in R.Q \setminus \{q_0\}$ is not greater than the repetition vector (cf. Definition 4.11), i.e., $q_{def} = q_{def} \mod \pi_{A_I \cup A_O}(n_{rep})$, and has a maximal number of output actor firings (cf. Equation (4.8)). To prove that $q_{def} \in Q_\gamma$, it remains to be shown that $q_{def}$ has a minimal number of input actor firings. For the states $Q_{ini}$ generated by Equation (4.12), this is trivially true. Hence, $\eta_1$ and $\eta_2$ in Equations (4.13) to (4.14) can be assumed to be minimal. Furthermore, it is self-evident that $state_{dep}(\max(\eta_1, \eta_2)) \geq state_{dep}(\max(\eta_1, \eta_2)) \geq state_{dep}(\eta_2)$. Thus, a lower bound for the minimal number of input actor firings for $state_{dep}(\max(\eta_1, \eta_2))$ is $\min\{\eta \in \mathbb{N}^{[A_I \cup A_O]} \mid \eta \geq \eta_1 \land \eta \geq \eta_2\}$. However, this is exactly the definition of $\max(\eta_1, \eta_2)$. Therefore, $R.Q \subseteq Q_\gamma$ and, thus, $Q_\gamma \equiv R.Q$.

Step 3: Derivation of the Cluster FSM Transition Set

Finally, the transition set $R.T$ of the cluster FSM is derived by partially ordering the interleaved cluster state space $Q_{lfp}$ using the $\geq$-relation between two cluster states. The resulting partial order can be visualized by a Hasse diagram, e.g., Figure 4.37b for the cluster $g_{\gamma_{1,2,a}}$ depicted in Figure 4.36. The edges in this diagram will be used to derive the set of transitions for the cluster FSM, e.g., as shown in Figure 4.38. Each edge in a Hasse diagram for a cluster state set $Q$ corresponds to a so-called tightly ordered pair $(q_{src}, q_{dst}) \in Q^2$ where $q_{dst} > q_{src}$ and no other state $q' \in Q$ is between (cf. Equation (4.20)) these two states. An example of this is the tightly ordered pair $(q_5, q_8)$ depicted in Figure 4.37a, which is missing in Figure 4.37b due to the cluster state $q_{10}$ between the states $q_5$ and $q_8$. More formally, the cluster FSM transition set is derived from the interleaved cluster state space $Q_{lfp}$ as follows:

Definition 4.12 (Cluster FSM Transition Set [FKH+08*]). Given the interleaved cluster state space $Q_{lfp}$, its set of tightly ordered pairs $TO_{lfp}$ can be

---

There is no explicit definition in [FKH+08*], but it is contained in [FKH+08*] Step 3.3 where the cluster FSM transition set is determined.
4. Clustering

derived (cf. Equation (4.19)). Subsequently, a transition \( t \) is created (cf. Equation (4.15)) for each tightly ordered pair that starts (cf. Equation (4.16)) from a state in the cluster FSM state set \( \mathcal{R}.Q \). Thus, the source state \( t.q_{src} \) of the transition can simply be set to \( q_{src} \) (cf. Equation (4.17)). However, the destination state \( q_{dst} \) of the tightly ordered pair \( (q_{src}, q_{dst}) \) is not necessarily contained in the cluster FSM state set \( \mathcal{R}.Q \). Hence, an equivalent state \( (\text{mod} \, \eta^c) \) has to be computed (cf. Equation (4.18)). More precisely, the cluster FSM transition set is defined as follows:

\[
\mathcal{R}.T = \{ t \mid (q_{src}, q_{dst}) \in \mathcal{T}_\mathcal{O}\lfp \}
\]
\[
\mathcal{T}_\mathcal{O}\lfp = \{ (q_{src}, q_{dst}) \in Q_{\mathcal{L}\mathcal{F}\mathcal{P}}^2 \mid q_{dst} > q_{src} \land \exists q' : q_{dst} > q' > q_{src} \}
\]

As can be seen from Definition 4.11, only a subset \( \mathcal{R}.Q \subset Q_{\mathcal{L}\mathcal{F}\mathcal{P}} \), which is depicted as dashed grouping in Figures 4.37 and 4.39, is used for the cluster FSM state set. Thus, not all tightly ordered pairs of \( \mathcal{T}_\mathcal{O}\lfp \) are required to construct the cluster FSM. To exemplify, the tightly ordered pairs of interest have been depicted in Figure 4.39a. These are the tightly ordered pairs that start from a state in the dashed grouping.

\( \textbf{(a) Hasse diagram with the relevant tightly ordered pairs for derivation of the cluster FSM transition set} \)

\( \textbf{(b) Cluster FSM state (boxes) and transition (directed edges) sets for subcluster } g_{\gamma_{1,2,3}} \)

**Figure 4.39:** Example for the derivation of the cluster FSM shown in Figure 4.38 from the interleaved cluster state space \( Q_{\mathcal{L}\mathcal{F}\mathcal{P}} \) of the subcluster \( g_{\gamma_{1,2,3}} \) depicted in Figure 4.36.

\( ^{61} \)The notation \( \mathcal{T}_\mathcal{O}_x \) is used to denote the set of tightly ordered pairs of a set of states \( Q_x \).
However, not all tightly ordered pairs starting in the dashed grouping also end in it, e.g., the tightly ordered pairs $(q_5, q_{10})$, $(q_3, q_7)$, $(q_2, q_7)$, and $(q_4, q_9)$ shown in Figure 4.39a. These cases correspond to those transitions that cross the boundary from one iteration of the cluster to the next one. Hence, an equivalent state contained in the cluster FSM state set has to be determined (cf. Equation (4.18)). For the destination states $q_{10}$, $q_7$, and $q_9$, it can be observed that subtracting the repetition vector $\eta_{\gamma_1,2,3}^{\text{rep}} = (1, 1, 1)$ will derive the equivalent states $q_3$, $q_1$, and $q_2$, respectively. These states will be present in the cluster FSM state set due to Equation (4.14). The presence of the original states $q_{\text{det}} \in Q_{\text{fp}} \setminus R, Q$ required for the existence of the tightly ordered pairs $(q_5, q_{10})$, $(q_3, q_7)$, $(q_2, q_7)$, and $(q_4, q_9)$ is ensured by Equations (4.4) and (4.13). The resulting transitions $q_5 \rightarrow q_3$, $q_3 \rightarrow q_1$, $q_2 \rightarrow q_1$, and $q_4 \rightarrow q_2$ for the tightly ordered pairs $(q_5, q_{10})$, $(q_3, q_7)$, $(q_2, q_7)$, and $(q_4, q_9)$ can be observed in Figure 4.39b.

To finish the construction of the cluster FSM, the actions $f(\ldots)$ and input/output guards $k_{\text{io}}$ have to be determined for each transition. This step, presented in Section 4.4.4, is independent of the clustering approach taken. Hence, it can be used both for the automata-based clustering approach detailed in this section or the rule-based approach given in the next section. The resulting cluster FSM for the composite actor $a_{\gamma_1,2,3}$ is depicted in Figure 4.38.

### 4.4.3 Rule-Based Clustering

The automata-based clustering methodology presented in the previous section may suffer from the state space explosions problem. This state space explosion can have two different underlying reasons: (1) the set of initial input/output dependency states $Q_{\text{ini}}$ is small, but computation of all possible interleavings by Definition 4.10 results in state space explosion, and (2) the state space explosion problem that is inherent in the marked graph expansion of a static DFG resulting in a possibly large initial set $Q_{\text{ini}}$. Note that the set of initial input/output dependency states might not suffer state space explosion even if the repetition vector sum is large. This is the case when the sequences of statically scheduled actor firings executed by the transitions of the cluster FSM are relatively large and, hence, only a small number of transitions are required to cover all actor firings of the repetition vector, even if the repetition vector sum is large. However, this does not mean that the computation of all possible interleavings does not produce a state space explosion. To ameliorate this reason for state space explosion, the rule-based approach published in [FZHT13*] will be presented in this section. This approach implicitly constructs the state

---

The marked graph expansion contains as many vertices as the repetition vector sum $\sum_{n \in I(\eta_{\gamma}^{\text{rep}})} \eta_{\gamma}^{\text{rep}}(n)$ of all the entries of the repetition vector $\eta_{\gamma}^{\text{rep}}$ of the cluster $g_{\gamma}$. 

---

145
4. Clustering

space $Q_\gamma$ of a cluster $g_\gamma$ using so-called rules. Note that the rule-based representation might achieve a more compact code size of a QSS as compared to the automata-based representation. However, if code size is not a concern, then the automata-based representation can achieve better overall overhead reduction due to its more efficient ability to select an enabled transition for execution as compared to selecting an enabled rule.

**Implicit Cluster State Space Representation**

In order to avoid the explicit enumeration of the cluster state space $Q_\gamma$, rules are proposed to define valid state transitions implicitly. At a glance, a rule $r \in R$ maps a subspace of the vector space $Q_{rules} = \mathbb{N}_{0,\infty}^{|A|}$ to a vector of actor firings $\eta$ (in the following called a partial repetition vector) that can be performed if the current state is contained in this subspace. More formally, a rule $r$ is defined as follows:

**Definition 4.13 (Rule [FZHT13*])**. A rule is a tuple $r = (l, u, \eta)$. The interval boundary vectors $l, u \in Q_{rules}$ define the state set $Q_r$ where the rule is active, i.e., $Q_r = \{ q \in Q_{rules} \mid l \leq q \leq u \}$. The partial repetition vector $\eta \in \mathbb{N}_{0,\infty}$ specifies how many firings to perform for each actor of the cluster when the rule $r$ is applied in a state $q_{cur} \in Q_r$.

Note that two different definitions are used for $Q_{rules}$. Initially, the algorithm works with lower and upper bounds $l, u \in Q_{rules} = \mathbb{N}_{0,\infty}^{|A|}$. Later, after rule merging has been applied, these bounds can be defined in terms of input actor firings, i.e., $l, u \in Q_{rules} = \mathbb{N}_{0,\infty}^{|A|} \cup \{ q_0 \}$. A detailed discussion of this is given in the rule merging subsection.

If a rule $r$ is applied in a current state $q_{cur} = q_{src}$, then the next state $q_{next}$ is calculated in two steps. First, the actor firings specified by $\eta$ are added to the source state (cf. Equation (4.21)) to compute a destination state. Then, the next state $q_{next}$ is the smallest equivalent state (cf. Equation (4.22)) of the destination state:

\[
q_{dst} = q_{src} + \pi_I(q_{src})(\eta) \quad (4.21)
\]
\[
q_{next} = q_{dst} \mod \pi_I(q_{dst})(\eta_{rep}) \quad (4.22)
\]

In the following, the algorithm that computes the set of rules $R_{fin}$ for the composite actor $a_\gamma \in \bar{g}.A$ that is derived via the cluster refinement operator $\xi_{rules}$, i.e., $\xi_{rules}(g, g_\gamma) = (\bar{g}, a_\gamma)$, will be reviewed. This set of rules implements a QSS for the actors $g_\gamma.A$ contained in the static subcluster $g_\gamma$ of the original DFG $g$.

---

63Incidentally, this requires the projection $\pi_I(q)(\ldots)$ in the Equations (4.21) and (4.22) to the index set $I(q)$ currently used by the state space $Q_{rules} \ni q$. 

146
The rule-based clustering algorithm works in three steps: (Step 1) computes the set of initial input/output dependency rules from the set of input/output dependency function values. (Step 2) derives the set of conflict-resolved rules that covers all necessary states and contains all necessary transitions of the cluster state space from the set of initial input/output dependency rules. (Step 3) obtains the set of final rules $R_{\text{fin}}$ where every rule fires at least one input actor from the set of conflict-resolved rules.

To avoid deadlocks, the worst case cluster environment has to be assumed and, thus, the resulting QSS always has to produce a maximum number of output tokens from a minimum number of input tokens. This criterion is satisfied for each output actor in isolation by constructing (Step 1) the set of initial input/output dependency rules $R_{\text{ini}}$ (cf. Definition 4.14) from the set of input/output dependency function values of the given output actor. The state space of the set of initial rules $R_{\text{ini}}$, i.e., the set of possible states for $q_{\text{cur}}$, in general, is only a subset of the cluster state space $Q_{\gamma}$. Furthermore, execution of the initial rules may perform transitions, i.e., state transitions as defined by Equations (4.21) to (4.22), that do not correspond to tightly ordered pairs in the state space of the rules $R_{\text{ini}}$.

To solve these irregularities, conflict resolution (cf. Definition 4.18) is performed in (Step 2) for each pair (cf. Definition 4.17) of conflicting (cf. Definition 4.16) rules. Conflict resolution will add new states to the state space of the set of rules until the state space of the resulting set of conflict-resolved rules is a superset of the cluster state space. Furthermore, conflict resolution will add additional rules that replace redundant rules (cf. Definition 4.19). Redundant rules are rules that do not correspond to tightly ordered pairs in the state space of the set of rules. These redundant rules can be discarded in the next step.

Finally, rule merging is performed in (Step 3). In general, the state space of the set of rules computed at the end of the previous step is a superset of the cluster state space. This is the case due to the presence of rules that do not fire any input actors. Hence, in general, to perform the equivalent of a transition as defined in Definition 4.12, a sequence of rule applications $\rho = \langle r_1, r_2, r_3, \ldots, r_n \rangle$ has to be taken. The first rule $r_1$ in this sequence will fire at least one input actor, while the rest of the sequence of rule applications $\langle r_2, r_3, \ldots, r_n \rangle$ will not contain any input actor firings. For each such sequence $\rho$, rule merging adds a rule that atomically fires all actor firings performed by this sequence of rule

\[ ^{64}\text{In general, the set of initial input/output dependency rules only covers a subset of the } \text{cluster state space} \ Q_\gamma \equiv R.Q \text{ and is also missing rules to cover all transitions } R.T. \]

\[ ^{65}\text{Of course, the exception of the rule that is active in the initial state } q_0 \text{ still applies. Remember that the initial state may not necessarily have maximized output actor firings. In this case, exactly one rule with no input actor firings is present in } R_{\text{fin}}. \]

\[ ^{66}\text{In contrast, the set of conflict-resolved rules may contain rules that fire only output or intermediate actors.} \]
4. Clustering

applications. After this operation, the state space of the set of final rules \( R_{\text{fin}} \) derived via the rule merging operation corresponds to the cluster state space as given in Definition 4.6. More precisely, the set of final rules \( R_{\text{fin}} \) for a consistent and deadlock-free subcluster \( g_\gamma \) is derived by the three steps given below:

**Step 1: Derivation of Initial Rules**

A rule \( r \) can be constructed from any two states \( q_{\text{src}}, q_{\text{dst}} \in Q_{\text{rules}} = \mathbb{N}_{0,\infty}^{\lvert A \rvert} \) with \( q_{\text{dst}} > q_{\text{src}} \). Then, the rule \( r \) generated by these two states has to perform \( r.\eta = q_{\text{dst}} - q_{\text{src}} \) actor firings (cf. Equation (4.24)). The lower bound \( r.l \) is equal to \( q_{\text{src}} \) (cf. Equation (4.25)), representing the minimum number of actor firings which have to be performed in order to enable the rule. If an actor \( a \) is fired by \( r \), i.e., \( \eta(a) > 0 \), the upper bound is set to the lower bound (cf. Equation (4.26)). This constrains the number of firings of the actor \( a \) to an interval which only contains the single value \( l(a) \). This requirement is due to the fact that only sufficient tokens can be assumed to execute \( \eta(a) \) firings starting from the \( l(a) \)’th firing of the actor \( a \). If an actor \( a \) is not fired by \( r \), only a minimum number of actor firings are required, i.e., \( u(a) = \infty \). This ensures that at least the minimum number of tokens have been produced by the actor \( a \). Note that if more firings of the actor \( a \) have already been performed than indicated by \( l(a) \), then the actor \( a \) must also have produced more tokens than required, not less.

The first step of the rule-based clustering approach is to find a set of initial input/output dependency rules \( R_{\text{ini}} \) based on the set of input/output dependency function values. More formally, the set of initial input/output dependency rules \( R_{\text{ini}} \) can be defined as given below:

**Definition 4.14** (The Set of Initial Rules [FZHT13*]). Given a static cluster \( g_\gamma \) and its set of input/output dependency function values \( H_{\text{dep}}(a_{\text{out}}) \), then its set of initial input/output dependency rules \( R_{\text{ini}} \) can be derived as follows:

\[
R_{\text{ini}} = \bigcup_{a_{\text{out}} \in A_{O}} \{ r \mid (q_{\text{src}}, q_{\text{dst}}) \in \mathcal{T}O_{\text{ini}}(a_{\text{out}}) \} \quad (4.23)
\]

\[
r.\eta = q_{\text{dst}} - q_{\text{src}} \quad (4.24)
\]

\[
r.l = q_{\text{src}} \quad (4.25)
\]

\[
r.u(a) = \begin{cases} r.l(a) & \text{if } r.\eta(a) > 0 \\ \infty & \text{otherwise} \end{cases} \quad (4.26)
\]

\[
\mathcal{T}O_{\text{ini}}(a_{\text{out}}) = \{ (q_{\text{src}}, q_{\text{dst}}) \in Q_{\text{ini}}(a_{\text{out}})^2 \mid q_{\text{dst}} > q_{\text{src}} \wedge \not\exists q' \in Q_{\text{ini}}(a_{\text{out}}) : q_{\text{dst}} > q' > q_{\text{src}} \} \quad (4.27)
\]

\[
Q_{\text{ini}}(a_{\text{out}}) = \{ \text{firings} \circ \text{state}_{\text{dep}}(\eta) \mid \eta \in H_{\text{dep}}(a_{\text{out}}) \} \cup \{ q_0 \} \quad (4.28)
\]
4.4 Cluster Refinement in Dynamic Data Flow Graphs

The construction of the set of initial input/output dependency rules proceeds by iterating over each output actor $a_{out} \in A_O$ (cf. Equation (4.23)) and computing a set of initial input/output dependency states $Q_{ini}(a_{out})$ (cf. Equation (4.28)) from the set of input/output dependency function values $H_{dep}(a_{out})$ of the selected output actor. From this set $Q_{ini}(a_{out})$, a set of tightly ordered pairs $\mathcal{T}\mathcal{O}_{ini}(a_{out})$ can be derived (cf. Equation (4.27)). Finally, for each tightly ordered pair $(q_{src}, q_{dst}) \in \mathcal{T}\mathcal{O}_{ini}(a_{out})$ (cf. Equation (4.23)), a rule is constructed (cf. Equations (4.24) to (4.26)) as described above and added to the set of initial input/output dependency rules $R_{ini}$. Note that each rule conforms to the rule property as given below:

**Definition 4.15** (Rule Property [FZHT13*]). A rule $r$ obeys the rule property if and only if $r.\eta(a) = 0$, then $r.u(a) = \infty$, and otherwise $r.l(a) = r.u(a)$.

In other words, for a rule $r$ to be well formed, actors which are not fired by the rule $r$ may have an arbitrary amount of firings, as long as the lower bound $r.l(a)$ is satisfied, while actors which are fired by rule $r$ must have a concrete number of firings $r.l(a) = r.u(a)$. This is trivially true for the initial input/output dependency rules as described in Section 4.4.3 as well as rules derived by conflict resolution in Definition 4.17.

Furthermore, according to Lemma 4.3 on page 165, any set of rules that has the property that all initial input/output dependency states are reachable from the initial state $q_0$ via application of these rules is a valid set of initial input/output dependency rules. Moreover, each set of initial input/output dependency states $Q_{ini}(a_{out})$ represents a chain, i.e., a totally ordered subset, of the partially ordered state space $Q_{rules}$. Hence, all initial input/output dependency states are reachable via application of rules from $R_{ini}$ starting from $q_0$, i.e., $R_{ini}$ is a valid set of initial input/output dependency rules.

To exemplify, consider the DFG given in Figure 4.36 on page 137, with the subcluster $g_{1,2,3}$ induced by the set of static actors $A_S = \{a_1, a_2, a_3\}$. The chains used to cover the set of initial input/output dependency states of $g_{1,2,3}$ (cf. Table 4.2 on page 140) is shown in Figure 4.40a. The initial rule $r_1$ corresponding to the edge $q_0 \rightarrow q_2$ is then derived as follows: $\eta = q_2 - q_0 = (1, 1, 0)$, $l = q_0 = (0, 0, 0)$, and $u = (0, 0, \infty)$. Therefore, in order to enable the rule, actors $a_1$ and $a_2$ must not have been fired before. Further initial input/output dependency rules are tabulated in Figure 4.40b.

Simulating the initial input/output dependency rules $R_{ini}$ as given in Figure 4.40b results in the FSM depicted in Figure 4.41a. It can be observed that all states of the FSM shown in Figure 4.38 on page 142 are present. However, the transitions are different, and not all states have outgoing transitions, therefore causing a deadlock if the FSM enters one of these states. These deadlocks

---

67 The ‘∘’ operator, e.g., $\text{firings} \circ \text{state}_{dep}$, is used to denote function composition, i.e., $(\text{firings} \circ \text{state}_{dep})(\eta) \equiv \text{firings}(\text{state}_{dep}(\eta))$. 

149
4. Clustering

Figure 4.40: Depicted are the initial input/output dependency states for the cluster \( g_{1,2,3} \). Three chains (cf. Table 4.2 on page 140) are used to cover all states. Each tightly ordered pair of a chain corresponds to an edge. The derived rules for the tightly ordered pairs are annotated to the edges. The corresponding set of initial input/output dependency rules is given in the table to the right.

(a) Input/output dependency state chains

\[
\begin{align*}
r_1 &= ((0,0,0), (0,1,1), (0,1,0)) \\
r_2 &= ((0,0,0), (0,1,0), (0,1,0)) \\
r_3 &= ((0,0,0), (0,1,0), (0,1,1)) \\
r_4 &= ((0,1,0), (0,1,0), (0,1,0)) \\
r_5 &= ((0,1,0), (0,1,1), (0,1,1)) \\
\end{align*}
\]

(b) The set of initial input/output dependency rules \( R_{ini} \)

\[
\begin{array}{|c|c|c|c|}
\hline
r & l & u & \eta \\
\hline
r_1 & (0,0,0) & (0,0,\infty) & (1,1,0) \\
r_2 & (0,0,0) & (\infty,0,\infty) & (0,1,0) \\
r_3 & (0,0,0) & (\infty,0,0) & (0,1,1) \\
r_4 & (1,1,0) & (1,\infty,\infty) & (1,0,0) \\
r_5 & (0,1,1) & (\infty,\infty,1) & (0,0,1) \\
\hline
\end{array}
\]

Figure 4.41: Shown are the conflict graph of the initial input/output dependency rules from Figure 4.40b and an FSM derived from simulating these rules. Applied rules as well as the corresponding sequence of statically scheduled actor firings are annotated to the transitions of the FSM. If two rules conflict, they are connected with an edge in the conflict graph shown to the right.

(a) FSM derived from simulating rules \( R_{ini} \)

(b) Conflict graph for initial input/output dependency rules \( R_{ini} \)

are caused by so-called conflict rules. More formally, the conflict relation for two rules is defined below:

Definition 4.16 (Rule Conflict [FZHT13*]). Given two rules \( r_1 \) and \( r_2 \), then these rules conflict, i.e., \( conflict(r_1, r_2) \), if and only if these rules have a common state \( q_{cur} \in Q_1 \cap Q_2 \) (cf. Equation (4.29)), and they fire at least one common actor (cf. Equation (4.30)).
4.4 Cluster Refinement in Dynamic Data Flow Graphs

\[ \text{conflict}(r_1, r_2) = Q_1 \cap Q_2 \neq \emptyset \quad (4.29) \]
\[ \land \min(r_1, \eta, r_2, \eta) \neq 0 \quad (4.30) \]

If two rules \( r_1 \) and \( r_2 \) conflict and are both enabled in a state \( q_{\text{cur}} \in Q_1 \cap Q_2 \), then applying one of them will disable the other rule.

**Proof.** [FZHT13*] Let actor \( a \) be such a common actor which is fired by both rules. Then, by construction (cf. Equation (4.26)), the lower and upper bounds of the corresponding intervals are as follows: \( r_1.I(a) = r_1.u(a) = n \) and \( r_2.I(a) = r_2.u(a) = m \). By the rule conflict definition, both rules have at least one common state. Hence, the intersection of their intervals is not empty. However, as both intervals contain only a single value, this can only be the case if \( n = m \). Let \( q_{\text{cur}}(a) = n \) in order to enable both \( r_1 \) and \( r_2 \). Without loss of generality, assume that \( r_1 \) is applied first. Then, for the next state \( q_{\text{next}}(a) = n + r_1.\eta(a) \) must hold. Note that \( r_1.\eta(a) > 0 \), as actor \( a \) is fired by \( r_1 \). However, as \( q_{\text{next}}(a) \neq m \) the rule \( r_2 \) is disabled. Analogously, if \( r_2 \) is fired first, \( r_1 \) will be disabled.

Note that if two rules \( r_1 \) and \( r_2 \) have no common states, they cannot conflict, as they can never be enabled at the same time. Also, if two rules have indeed a common state \( q_{\text{cur}} \), but do not fire common actors, they cannot conflict either.

**Proof.** [FZHT13*] Without loss of generality, let actor \( a \) be such an actor which is fired by \( r_1 \) but not by \( r_2 \). Then, by construction, the lower and upper bounds of the corresponding intervals are as follows: \( r_1.I(a) = r_1.u(a) = n \) and \( r_2.I(a) = r_2.u(a) = \infty \). By the rule conflict definition, both rules have at least one common state. Hence, the intersection of their intervals is not empty. This can only be the case if \( n \geq m \). Let \( q_{\text{cur}}(a) = n \) in order to enable both \( r_1 \) and \( r_2 \). If \( r_1 \) is fired first, it follows for the next state that \( q_{\text{next}}(a) = n + r_1.\eta(a) > n \geq m \), i.e., \( r_2 \) is still enabled. If, on the other hand, \( r_2 \) is applied first, it follows for the next state that \( q_{\text{next}}(a) = n + r_2.\eta(a) = n \), i.e., \( r_1 \) remains enabled.

An example of conflict rules are \( r_1 \) and \( r_3 \) from the set of initial input/output dependency rules \( R_{\text{ini}} \) given in Figure 4.40b. Both rules \( r_1 \) and \( r_3 \) are enabled in state \( q_0 \). However, if \( r_1 \) is fired first, \( r_3 \) cannot be fired anymore. On the other hand, if \( r_3 \) is fired first, \( r_1 \) cannot be fired anymore. These conflicts will be resolved next.

**Step 2: Resolving Conflict Rules**

In order to resolve these conflicts, new rules will be added to the set of initial input/output dependency rules \( R_{\text{ini}} \) by a least fixed-point operation to form the set of conflict-resolved rules \( R_{\text{lfp}} \). For each pair \( r_1 \) and \( r_2 \) of conflicting rules,
common actor firings are extracted from both rules, resulting, in the general case, in two additional rules \( r_1^1 \) and \( r_2^1 \). Then, \( r_1^1 \) can be applied after \( r_1 \), and analogously, \( r_1^1 \) can be executed after \( r_2 \) (For a detailed proof cf. Section 4.4.5 Lemma 4.2 on page 165). More formally, the conflict resolution operator is defined as follows:

**Definition 4.17** (Conflict Resolution Operator [FZHT13*]). Given two conflicting rules \( r_1 \) and \( r_2 \), the conflict resolving operation \((r_1^1, r_2^1) = r_1 \circ r_2\) generates two new rules \( r_1^1 \) and \( r_2^1 \) such that \( r_1^1 \) can be executed after \( r_1 \), and \( r_2^1 \) can be executed after \( r_2 \). Let \( \eta_c = \min(r_1 \cdot \eta, r_2 \cdot \eta) \) be the pointwise minimum of the partial repetition vectors \( r_1 \cdot \eta \) and \( r_2 \cdot \eta \). Note that \( \eta_c \) represents the common actor firings of \( r_1 \) and \( r_2 \). Then, the lower bounds and partial repetition vectors of \( r_1^1 \) and \( r_2^1 \) can be calculated as follows:

\[
\begin{align*}
 r_1^1 \cdot l &= r_1 \cdot l + \eta_c \\
 r_2^1 \cdot l &= r_2 \cdot l + \eta_c
\end{align*}
\]

The upper bounds \( r_1^1 \cdot u \) and \( r_2^1 \cdot u \) are then calculated from these lower bounds and partial repetition vectors as described by Equation (4.26).

An upper bound for the possible state space of \( q_{\text{cur}} \) can be determined from Equation (4.22) as \( q_{\text{cur}} \in Q_{\text{cur}} = \{ q \in Q_{\text{rules}} \mid 0 \leq q \nless \eta_{\gamma}^{\text{rep}} \} \). Note that, in the general case, the conflict resolution operator ‘\( \circ \)’ might produce rules \( r_1^1 \) and \( r_2^1 \) with lower bounds greater than or equal to the repetition vector of the cluster. However, such rules can never be executed since the current state \( q_{\text{cur}} \) will never be greater than or equal to the repetition vector. Note that such rules must not be discarded, but must be shifted, i.e., \( r' = r + k \cdot \eta_{\gamma}^{\text{rep}} \), to their canonical position in the state space.\(^{68}\)

The canonical position for an arbitrary state \( q \in Q_{\text{rules}} \) is \( q' = q \mod \eta_{\gamma}^{\text{rep}} \), i.e., the state \( q' = q + k \cdot \eta_{\gamma}^{\text{rep}}, k \in \mathbb{Z} \) such that \( 0 \leq q' \nless \eta_{\gamma}^{\text{rep}} \). Equivalently, the notation \( r \mod \eta \) is used to denote a modulo operation for rules.\(^{69}\) The modulo operation \( r' = r \mod \eta_{\gamma}^{\text{rep}} \) shifts the state space \( Q_r \) where the rule \( r \) is applicable into the new state space \( Q_{r'} \) where the new rule \( r' \) is applicable by adding an integer multiple of the repetition vector to each state in \( Q_r \) in such a way that the intersection \( Q_{\text{cur}} \cap Q_{r'} \) is not empty.

---

68 The notation \( r' = r + \eta_{\text{shift}} \) is used to denote that the rule \( r' \) is a copy of \( r \) where the state space \( Q_{r'} \) has been derived from shifting the state space \( Q_r \) by adding the partial repetition vector \( \eta_{\text{shift}} \) to each state in \( Q_r \), i.e., \( r' \cdot l = r \cdot l + \eta_{\text{shift}}, r' \cdot u = r \cdot u + \eta_{\text{shift}}, \) and \( r' \cdot \eta = r \cdot \eta.\)

69 The notation \( r' = r \mod \eta \) is used to denote a modulo operation for rules. The modulo operation \( \mod \) is defined for rules \( r.l, r.u \in \mathbb{Z} \) and \( \eta \in \mathbb{N}^d \). It computes the shifted rule \( r' = r + k \cdot \eta, k \in \mathbb{Z} \) such that \( r' \cdot l \) is the smallest vector with \( r' \cdot l \geq 0.\)
Moreover, since rules may now be shifted (cf. Equation (4.35)), the conflict check (cf. Equation (4.38)) must now also detect conflicts between shifted versions (cf. Equation (4.37)) of rules. More formally, the set of conflict-resolved rules $R_{\text{lfp}}$ for a cluster can be derived as follows:

**Definition 4.18** (The Set of Conflict-Resolved Rules [FZHT13*]). Given the set of initial input/output dependency rules $R_{\text{ini}}$, then the set of conflict-resolved rules $R_{\text{lfp}}$ is derived by computing the least fixed point (cf. Equation (4.31)) of the resolve function. The resolve function is a monotonically increasing function that enlarges (cf. Equation (4.33)) the set of rules $R$ starting (cf. Equation (4.34)) from the set of initial input/output dependency rules $R_{\text{ini}}$ by adding (cf. Equation (4.35)) the canonical version of all rules $r_1$ and $r_2$ generated by the conflict resolution operator $\odot$ (cf. Equation (4.39)) for all pairs (cf. Equation (4.36)) of possibly shifted (cf. Equation (4.37)) conflicting rules (cf. Equation (4.38)). This can be defined precisely as follows:

\[ R_{\text{lfp}} = \text{lfp}(R = \{ r \in \text{resolve}(R) \mid r.\eta > 0 \}) \]
\[ \text{resolve}(R) = R \cup R_{\text{ini}} \cup \{ r_1 \mod \eta_{\text{rep}} \gamma, r_2 \mod \eta_{\text{rep}} \gamma \mid \exists r_1, r_2 \in R, \] \[ n \in \mathbb{Z}, r_2' = r_2 + n \cdot \eta_{\text{rep}} \gamma : \] \[ \text{conflict}(r_1, r_2') \] \[ (r_1, r_2') = r_1 \odot r_2' \} \]

(4.31) \hspace{2cm} (4.32) \hspace{2cm} (4.33) \hspace{2cm} (4.34) \hspace{2cm} (4.35) \hspace{2cm} (4.36) \hspace{2cm} (4.37) \hspace{2cm} (4.38) \hspace{2cm} (4.39)

Note that the conflict resolution operator $\odot$ might also produce rules with no actor firings. These rules can be discarded (cf. Equation (4.32)). However, the production of such rules hints at the presence of redundant rules in the set of conflict-resolved rules $R_{\text{lfp}}$. A special case arises that leads to redundant rules during conflict resolution if for a pair of conflicting rules $r_{\text{other}}$ and $r_{\text{red}}$ the rule $r_{\text{other}}$ is more general than the rule $r_{\text{red}}$. More formally, the set of redundant rules $R_{\text{red}}$ can be derived as follows:

**Definition 4.19** (Rule Redundancy [FZHT13*]). Given a set of rules $R_{\text{lfp}}$, then the function $\text{redundant}(R_{\text{lfp}}) = R_{\text{red}}$ selects all rules in $R_{\text{lfp}}$ that are redundant due to the presence of another rule in $R_{\text{lfp}}$. In detail, a rule $r_{\text{red}}$ is redundant if and only if another rule $r_{\text{other}}$ exists (cf. Equation (4.40)) in $R_{\text{lfp}}$ that can be applied in a superset $Q_{\text{other}} \supseteq Q_{\text{red}}$ of states where the redundant
rule \( r_{\text{red}} \) can be applied (cf. Equation (4.41)) and the other rule fires a subset of the actor firings of the redundant rule (cf. Equation (4.42)).

\[
\text{redundant}(R_{\text{dfp}}) = \{ r_{\text{red}} \in R_{\text{dfp}} \mid \exists r_{\text{other}} \in R_{\text{dfp}} \setminus \{ r_{\text{red}} \} : \\
\begin{align*}
\eta_{\text{other}} \cdot l & \leq r_{\text{red}} \cdot l \land r_{\text{other}} \cdot u \geq r_{\text{red}} \cdot u \land \\
r_{\text{red}} \cdot \eta & \geq r_{\text{other}}
\end{align*}
\] (4.40)

Conflict resolution of \( r_{\text{red}} \) and \( r_{\text{other}} \) will have generated rules \( r_{\text{other}}^1 \) and \( r_{\text{red}}^1 \). In this case, the common actor firings are \( \eta_c = \min(\eta_{\text{other}}, \eta_{\text{red}}, \eta) = \eta_{\text{other}} \cdot \eta \), and thus, \( r_{\text{other}}^1 \cdot \eta = 0 \). This means that \( r_{\text{other}}^1 \) can be discarded (cf. Equation (4.32)) because it does not fire any actors. Also, \( r_{\text{red}} \) can be eliminated as a redundant rule.

**Proof.** [FZHT13*] Lemma 4.2 on page 165 shows that \( r_{\text{red}}^1 \) can be applied after \( r_{\text{other}} \) and will lead to the same state as firing \( r_{\text{red}} \) directly. Consequently, in all states where \( r_{\text{red}} \) can be applied, the sequence of rule applications \( (r_{\text{other}}, r_{\text{red}}^1) \) can also be taken, and will result in the same destination state. Hence, rule \( r_{\text{red}} \) is redundant.

To exemplify, consider the set of initial input/output dependency rules \( R_{\text{ini}} = \{ r_1, r_2, \ldots, r_5 \} \) from Figure 4.40b. The corresponding conflict graph is shown in Figure 4.41b. As can be seen in the conflict graph, the rules \( r_1 \) and \( r_2 \), \( r_1 \) and \( r_3 \), as well as \( r_2 \) and \( r_3 \) conflict.

In the first step, the conflict between \( r_1 \) and \( r_2 \) will be resolved. A common state of these rules is \( q_0 \). The vector of common actor firings is \( \eta_c = (0,1,0) \). Hence, the conflict resolving operation \( r_1 \circ r_2 \) results in two rules: \( r_1^1 = ((0,1,0), (0, \infty, \infty), (1,0,0)) \), and \( r_2^1 = ((0,1,0), (\infty, \infty, \infty), (0,0,0)) \) (which can be discarded due to the zero partial repetition vector). Note that \( r_2 \) is more general than \( r_1 \). Therefore, \( r_1 \) can be eliminated as a redundant rule. It should be noted that this conflict resolving operation also resolves the conflict between \( r_1 \) and \( r_3 \). The set of rules after this first conflict resolving step is \( R_{\text{ini}}^\prime = \{ r_1^1, r_2, r_3, r_4, r_5 \} \). The resulting FSM and conflict graph are depicted in Figure 4.42.

Analogously, the second step resolves the conflict between \( r_2 \) and \( r_3 \). The set of conflict-resolved rules after this second (and last) conflict resolving step is \( R_{\text{dfp}} \setminus R_{\text{red}} = \{ r_1^1, r_2, r_3^1, r_4, r_5 \} \). The resulting FSM and conflict graph are depicted in Figure 4.43.

Comparing the FSM in Figure 4.38 on page 142 to the FSM obtained by simulation in Figure 4.43a, the former features a transition from \( q_4 \) to \( q_7 \) firing both \( a_2 \) and \( a_3 \), while in the latter, this transition is split into two transitions: One transition from \( q_4 \) to \( q_7 \) firing \( a_3 \) only, and a second transition from \( q_7 \) to \( q_2 \) firing \( a_2 \). The reason for this can be found in the conflict resolution step:
Figure 4.42: Listed to the right is the set of rules $R_{ini}^1$ derived from one step of conflict resolution of the set of initial input/output dependency rules $R_{ini}$. The redundant rule $r_1$ has already been discarded.

Although it generates rules which guarantee a deadlock-free execution of the composite actor, it may also generate rules which do not fire any input actors.

While this is perfectly valid if counter variables for all actors are maintained during the execution of the composite actor, it becomes problematic if only counter variables for input actors are used, which is desirable, as it means less checks on rule conditions, and thus, less scheduling overhead for the composite actor. This is possible, as the state of the cluster can always be determined solely based on the number of input actor firings.

For example, consider rule $r_2$ in Figure 4.43b, which is only applicable if the counter for output actor $a_2$ is zero. However, eliminating the condition for $a_2$ results in a rule which is always applicable, as the firing intervals corresponding to $a_1$ and $a_3$ are not constrained.

In order to only use counter variables for input actors, all rules that do not fire any input actors must be eliminated. This is discussed next.

Step 3: Rule Merging

Let $r_{in}$ be a rule which fires at least one input actor (and possibly some output actors), and $r_{ni}$ be a rule which does not fire any input actors. After the rule $r_{in}$ has been applied, the set of possible states can be computed as $Q'_{in} = Q_{in} + r_{in} \eta$.\footnote{As always, the initial cluster state $q_0$ is the exception.} In principle, $r_{ni}$ can be applied after $r_{in}$ if the set of states $r_{ni}$

\[ r_1 = ((0,1,0),(0,\infty,\infty),(1,0,0)) \]
\[ r_2 = ((0,0,0),(\infty,0,0),(0,1,0)) \]
\[ r_3 = ((0,0,0),(\infty,0,0),(0,1,1)) \]
\[ r_4 = ((1,1,0),(1,\infty,\infty),(1,0,0)) \]
\[ r_5 = ((0,1,1),(\infty,\infty,1),(0,0,1)) \]
4. Clustering

\begin{figure}[h]
\centering
\begin{subfigure}{0.45\textwidth}
\centering
\begin{tikzpicture}
\node (q0) at (0,0) {$q_0$} ;
\node (q1) at (1,0) {$q_1$} ;
\node (q2) at (2,0) {$q_2$} ;
\node (q3) at (2,-1) {$q_3$} ;
\node (q4) at (1,-1) {$q_4$} ;
\node (q5) at (0,-1) {$q_5$} ;
\node (q6) at (0,-2) {$q_6$} ;
\node (q7) at (1,-2) {$q_7$} ;

\path (q0) edge node {$f_{(a_2)}$} (q2);
\path (q1) edge node {$f_{(a_3)}$} (q3);
\path (q2) edge node {$f_{(a_2)}$} (q1);
\path (q3) edge node {$f_{(a_1)}$} (q4);
\path (q4) edge node {$f_{(a_3)}$} (q5);
\path (q5) edge node {$f_{(a_2)}$} (q4);
\path (q6) edge node {$f_{(a_2)}$} (q1);
\path (q7) edge node {$f_{(a_2)}$} (q6);
\end{tikzpicture}
\end{subfigure}
\begin{subfigure}{0.45\textwidth}
\begin{itemize}
\item $r_1 = (0, 1, 0), (0, \infty, \infty), (1, 0, 0))$
\item $r_2 = (0, 0, 0), (\infty, 0, \infty), (0, 1, 0))$
\item $r_3 = (0, 1, 0), (\infty, \infty, 0), (0, 0, 1))$
\item $r_4 = (1, 1, 0), (1, \infty, \infty), (1, 0, 0))$
\item $r_5 = (0, 1, 1), (\infty, \infty, 1), (0, 0, 1))$
\end{itemize}
\end{subfigure}
\caption{Simulated FSM after second conflict resolving step}
\caption{Conflict graph for rules after second conflict resolving step}
\end{figure}

\textbf{Figure 4.43:} Shown to the right is the set of conflict-resolved rules after removal of the redundant rules, i.e., $R_{\text{dfp}} \setminus R_{\text{red}}$. The FSM derived from simulating these rules is depicted to the left.

$Q_c = Q'_m \cap Q_{ni}$ is non-empty (cf. Figure 4.44a). Then, $r_{ni}$ can be merged with $r_{ni}$, resulting in a new merged rule $r_m$, which combines the actor firings performed separately by $r_{ni}$ and $r_{ni}$, i.e., $r_m = r_{ni} \cdot \eta + r_{ni} \cdot \eta$ (cf. Figure 4.44b).

The state set $Q_m$ of this merged rule can simply be calculated by shifting back the state set $Q_c$, i.e., $Q_m = Q_c - r_{ni} \cdot \eta \subseteq Q_{in}$.

However, inserting such a merged rule $r_m$ into the conflict-free set of rules would destroy this property since $r_{ni}$ conflicts with $r_m$ due to common actor firings, i.e., $r_{ni} \cdot \eta \leq r_m \cdot \eta$, and common states, i.e., $Q_m \subseteq Q_{in}$. Therefore, the merge operation must also restrict the state set $Q_m$ of the original rule $r_m$ such that it can no longer be applied to states in $Q_m$ (where $r_m$ can be applied). To this end, the set $\hat{Q}_{in} = Q_{in} \setminus Q_m$ has to be calculated in terms of interval boundary vectors. Without going into detail, this operation results in a set of rules $\hat{R}_{in} = \{\hat{r}_{in,0}, \hat{r}_{in,1}, \ldots, \hat{r}_{in,n}\}$ necessary to cover all states in $Q_{in}$. The rules in $\hat{R}_{in}$ may not be conflict-free, but as they are derived from the same rule,

$q' = \{q + \eta_{\text{shift}} \mid q \in Q \}$. This operation can be performed by shifting the corresponding interval boundary vectors $l$ and $u$ of the set $Q$ by adding the partial repetition vector $\eta_{\text{shift}}$ to them.

Note that in this case, the interval boundary vectors $l_c$ and $u_c$ corresponding to $Q_c$ can be calculated by intersecting the interval boundary vectors of $Q_{ni}$ and $Q'_m$, i.e., $l_c = \max(r_{ni} \cdot l, r_{m} \cdot l + r_{m} \cdot \eta)$, and $u_c = \min(r_{m} \cdot u, r_{m} \cdot u + r_{m} \cdot \eta)$.
4.4 Cluster Refinement in Dynamic Data Flow Graphs

![Graphical depiction of a rule merging situation.](image)

**Figure 4.44:** Graphical depiction of a rule merging situation. Shown on the left side is the original situation. Depicted on the right side is the resulting situation after merging has been performed.

namely $r_{ni}$, they can be safely added to the set of final rules $R_{fin}$. Note that as $r_{ni}$ will be eliminated later, it does not have to be restricted like $r_{in}$.

An important point which has to be considered during the rule merging step is that if $r_{ni}$ is eliminated after the rule merging step, the merge operation has to be an over-approximation. Otherwise, some states in the FSM may not have outgoing transitions, therefore causing deadlocks during the execution of the composite actor. Given two rules $r_{in}$ and $r_{ni}, Q_c$ may be empty, in which case they will not be merged. As a result, when merging rules, equivalent rules of $r_{ni}$, i.e., rules $r_{ni} + k \cdot \eta^\text{rep}, k \in \mathbb{N}_0$ must also be considered. If, however, due to $k > 0$, the merged rule $r_m$ has $r_{m,l} \geq \eta^\text{rep}$, it can safely be discarded, as the counter variables maintained during simulation of the composite actor cannot reach the values of such lower bounds of the condition intervals.

Finally, a special rule may exist which fires only output actors, but must not be eliminated after the rule merging step: This is the case if sufficient initial tokens, e.g., channels $c_{1 \rightarrow 2}$ and $c_{3 \rightarrow 2}$ in the DFG $g_{ex2}$ depicted in Figure 4.36 on page 137, are contained in the cluster such that output actors can be fired without firing any input actors. Obviously, eliminating this special rule would prevent the composite actor from running at all as the initial state $q_0$ would have no outgoing transitions. As only one such rule can exist, it can easily be tagged as special and, thus, can be retained after the merging step.

To exemplify, the conflict-free set of rules $R_{conf} \setminus R_{red} = \{r_1, r_2, r_3, r_4, r_5\}$ from Figure 4.43b is considered. Rule $r_2$ fires only output actors, namely $a_2$. First, this rule is merged with rule $r^1$, which fires $a_1$, an input/output actor. For $k = 0$, $l_c = \max((0,0,0),(0,1,0) + (1,0,0)) = (1,1,0)$, and $u_c = \min((\infty,0,\infty),(0,\infty,\infty) + (1,0,0)) = (0,1,\infty)$. In this case, $r^1$ and $r_{2,0}$ cannot be merged, as $l_c \notin u_c$. 

157
4. Clustering

![Diagram of simulated final rules $R_{\text{fin}}$](image)

**Figure 4.45:** Depicted is the set of final rules $R_{\text{fin}}$ for subcluster $g_{1,2,3}$ from Figure 4.36. This set of rules has been derived from the set of conflict-free rules shown in Figure 4.43b via rule merging. The resulting FSM derived from state space expansion of the set of final rules $R_{\text{fin}}$ is also shown to the right. Applied rules are annotated to the transitions.

However, for $k = 1$, $l_c = \max((0, 0, 0) + (1, 1, 1), (0, 1, 0) + (1, 0, 0)) = (1, 1, 1)$ and $u_c = \min((\infty, 0, \infty) + (1, 1, 1), (0, \infty, \infty) + (1, 0, 0)) = (1, 1, \infty)$. Now, $l_c \leq u_c$, and $r_{1.2}$ and $r_{2.1}$ can be merged. The merged rule is calculated as $r_{1.2} = ((1, 1, 1) - (1, 0, 0), (1, 1, \infty) - (1, 0, 0), (1, 0, 0) + (0, 1, 0)) = ((0, 1, 1), (0, 1, \infty), (1, 1, 0))$. The set of restricted rules $R_{\text{fin}}^1$ contains only a single rule (after pruning another rule which could never be applied), namely $\hat{r}_{1.0}^1 = ((0, 1, 0), (0, \infty, 0), (1, 0, 0))$.

It can be observed that $r_{1.2}$ can be applied in state $q_5 = (0, 1, 2)$ (cf. Figure 4.45a), creating the missing (direct) transition from $q_5$ to $q_3$ (contrast Figure 4.43a to Figure 4.38 on page 142). Moreover, the restricted rule $r_{1.0}^1$ in the form of $\hat{r}_{1.0}^1$ can no longer be applied in $q_5$, thus eliminating state $q_6$.

Merging $r_2$ with the remaining states $r_{3.2}^1$, $r_2$, and $r_5$ results in the rules $R_{\text{fin}}$ given in Figure 4.45b. Note that $r_2$ has been retained as discussed above.

The final step of the rule-based clustering approach is to schedule the partial repetition vector of each rule $r \in R_{\text{fin}}$, the set of rules after the merging step.

### 4.4.4 Scheduling Static Sequences

In this section, the input/output guard $k^{io}$ and the action $f^{i..}$ are derived for each transition of the cluster FSM or each rule of the set of final rules that implement the QSS for the composite actor $a_\gamma$ derived via the $\xi_{FSM}$ or $\xi_{rules}$ operator, respectively. The actions execute sequences of stat-
ically scheduled actor firings for the actors $a \in g_\gamma$. To generate these sequences of statically scheduled actor firings, a partial repetition vector $\eta \in \mathbb{N}_0^{|A|}$ is required. For the rule-based approach, the partial repetition vector $r.\eta$ of a rule $r$ is readily available for all rules after rule merging has been performed, but before the state space has been reduced to only use input actors. For the automata-based approach, the partial repetition vector of a transition can be computed from the $\text{firings}$ function as well as the source state $q_{\text{src}}$ and the destination state $q_{\text{dst}}$ of the transition. More precisely, the partial repetition vector $t.\eta$ of a transition $t$ can be defined as follows:

$$t.\eta = \begin{cases} \eta^- \text{rep} \\ (\text{firings}(t.q_{\text{dst}}) - \text{firings}(t.q_{\text{src}})) \mod \eta^- \text{rep} \end{cases}$$

if $t.q_{\text{src}} = t.q_{\text{dst}}$ \hspace{1cm} (4.43)
otherwise

The $t.q_{\text{src}} = t.q_{\text{dst}}$ case in Equation (4.43) covers self-loops. A self-loop represents monolithic clustering, which executes a whole iteration of the static DFG contained in the cluster. Monolithic clustering is performed by the automata-based clustering algorithm if the $\xi_{SDV}$ cluster refinement operation is safe. For an exact definition of this safety property, refer to Theorem 4.1 on page 106. Note that the cluster FSM may also include a retiming sequence (cf. Figure 4.16d on page 108) before entering this self-loop state.

In order to generate the input/output guard $k^{io}$ for a transition or rule, knowledge of the cluster state in which the actor firings of the partial repetition vector will be performed is required. This is due to the possible presence of CSDF actors in the cluster $g_\gamma$. The communication behavior of a CSDF actor depends on the current phase of the actor. To determine which phases of the CSDF actors are executed by the partial repetition vector, the starting state is required. For the automata-based approach, the source state $t.q_{\text{src}}$ is used to determine the starting state. For the rule-based approach, the lower bound $r.l$ is used. With this information, the number of consumed and produced tokens on the cluster input and output ports by the execution of the sequences of statically scheduled actor firings derived from the partial repetition vector can be determined. Thus, the input/output guard $k^{io}$ for each transition or rule can be computed. More formally, using the function $\text{io} : Q_\gamma \rightarrow \mathbb{Z}^{[\text{I} \cup \text{O}]}$ to map from a cluster state to the number of consumed (negative) and produced (positive) tokens on the cluster input and outputs ports, the input/output guards for all transitions can be defined as follows:

$$\text{cons}(t.I) = -\pi_I(\text{io}(t.q_{\text{src}} + \pi_{A_I \cup A_O}(t.\eta)) - \text{io}(t.q_{\text{src}})) \hspace{1cm} \forall t \in T$$
$$\text{prod}(t.O) = \pi_O(\text{io}(t.q_{\text{src}} + \pi_{A_I \cup A_O}(t.\eta)) - \text{io}(t.q_{\text{src}})) \hspace{1cm} \forall t \in T$$

73Here, the input/output guard is defined indirectly via the notation $\text{cons}(t.I)$ and $\text{prod}(t.O)$ that denotes the number of consumed and produced tokens when the transition is taken.
4. Clustering

Equivalently, one can define the input/output guards for all rules:

\[ \text{cons}(r, I) = -\pi_I(\pi_{A_I \cup A_D}(r.l + r.\eta)) - \pi_I(\pi_{A_I \cup A_D}(r.l)) \quad \forall r \in R \]
\[ \text{prod}(r, O) = \pi_O(\pi_{A_I \cup A_D}(r.l + r.\eta)) - \pi_O(\pi_{A_I \cup A_D}(r.l)) \quad \forall r \in R \]

Finally, the partial repetition vectors \( t.\eta \) or \( r.\eta \) for all transitions or rules, respectively, are scheduled by a modified version of the cycle-breaking algorithm presented in [HB07], which, for pure SDF graphs, always finds an Single Appearance Schedule (SAS) if existent. In an SAS, each actor occurs only once, therefore minimizing code memory size when inlining the actor firings.

Basically, an SAS corresponds to a topological sorting of the actors contained in the static cluster. This is achieved by topologically sorting the Strongly Connected Components (SCCs) of the cluster. Subsequently, for each non-trivial SCC, the algorithm removes the edges with enough initial tokens to perform a whole iteration of the subgraph of the cluster induced by the SCC. If the SCC is still strongly connected after removing such edges, the subgraph is said to be tightly connected and is scheduled by simulation. Otherwise, the SCC is loosely connected, and the algorithm is applied recursively to this SCC.

For partial repetition vectors, however, it may not always be possible to find an SAS even if one exists. This is due to the fact that the partial repetition vector in question may not be a multiple of the repetition vector of the subgraph of the cluster induced by the SCC. The generated looped schedules implementing the sequences of statically scheduled actor firings are used as actions \( f(\ldots) \) by the software synthesis to generate C++ source code, which is itself compiled by gcc to generate the executable.

### 4.4.5 Proving Semantic Equivalence

Before presenting experimental results showing the benefits of the presented clustering algorithms, a formal proof of the correctness of the approach is given. The following proofs are based on infinite version \( Q_{\text{ini}}^\infty \) of the set of initial input/output dependency states \( Q_{\text{ini}} \).

The semantic equivalence (cf. Definition 4.2 on page 99) of the composite actor \( a_\gamma \) and the static DFG \( g_\gamma \) is proven in Theorem 4.5 by showing the equivalence of the denotational KPN descriptions [Kah74] \( K_{a_\gamma} \) and \( K_{g_\gamma} \), for both \( a_\gamma \) and \( g_\gamma \).
To exemplify the denotational description, the DFG from Figure 4.36 is used. For the sake of convenience, this DFG is replicated here as Figure 4.46.

---

\( ^{74} \)Likewise, the input/output guard of a rule is defined indirectly via the notation \( \text{cons}(r, I) \) and \( \text{prod}(r, O) \) that denotes the number of consumed and produced tokens when the rule is applied.

\( ^{75} \)For a given state space \( Q \), the infinite version is defined as \( Q^\infty = \{ q + n \cdot \pi_{\{q\}^\infty}(\eta_\gamma^\text{rep}) \mid q \in Q, n \in \mathbb{N}_0 \} \).
denotational description $\mathcal{K}_g\gamma$ maps tuples $s_{in} = (s_{i1}, s_{i2})$ of sequences $s \in \mathcal{V}^{**}$ of tokens $\bullet$ on its input ports $i_1$ and $i_2$ to tuples $\mathcal{K}_g\gamma(s_{in}) = s_{out} = (s_{o1}, s_{o2}, s_{o3})$ of sequences of tokens on its output ports $o_1$, $o_2$, and $o_3$. As can be seen from the example values of $\mathcal{K}_g\gamma$ given below, the denotational description naturally expresses the maximum output from input property, which should be preserved by the clustering algorithm.

$$\mathcal{K}_g\gamma(\lambda, \lambda) = (\lambda, (\bullet), \lambda)$$

$$\mathcal{K}_g\gamma((\bullet), \lambda) = ((\bullet), (\bullet), (\bullet))$$

$$\mathcal{K}_g\gamma((\bullet , \bullet), \lambda) = ((\bullet), (\bullet), (\bullet))$$

$$\mathcal{K}_g\gamma((\bullet, \bullet , \bullet , \cdots), \lambda) = ((\bullet), (\bullet), (\bullet))$$

$$\mathcal{K}_g\gamma((\bullet, \bullet), (\bullet)) = ((\bullet), (\bullet), (\bullet))$$

$$\mathcal{K}_g\gamma((\bullet, \bullet , \bullet), (\bullet)) = ((\bullet), (\bullet), (\bullet))$$

$$\mathcal{K}_g\gamma((\bullet, \bullet , \bullet , \cdots), (\bullet)) = ((\bullet), (\bullet), (\bullet))$$
Note that there are an infinite number of tuples $s_{in}$ and $s_{out}$ representing the infinite stream of tokens that is processed by a cluster. Hence, later on, a mathematical proof for the equivalence of $K_{g_\gamma}$ and $K_{a_\gamma}$ will be given. For this proof, the token values can be abstracted away by using $\bullet$ as token symbol. This can be done because clustering can also be thought of as KPN process composition. Therefore, the same operations will always be performed on the same values. However, it is not trivially obvious if the set of rules generated by the clustering algorithm will always produce the maximum number of output tokens from the given input tokens. For example, a fully static scheduling of $g_\gamma$ by the composite actor $a_\gamma$ would produce the following KPN function $K_{a_\gamma}$, which is not equivalent to $K_{g_\gamma}$:

$$K_{a_\gamma}(\lambda, \lambda) = (\lambda, \lambda, \lambda)$$

$$K_{a_\gamma}((\bullet), \lambda) = (\lambda, \lambda, \lambda)$$

$$K_{a_\gamma}((\bullet, \ldots), \lambda) = (\lambda, \lambda, \lambda)$$

$$K_{a_\gamma}(\lambda, (\bullet)) = (\lambda, \lambda, \lambda)$$

$$K_{a_\gamma}(\lambda, (\bullet, \ldots)) = (\lambda, \lambda, \lambda)$$

$$K_{a_\gamma}((\bullet), (\bullet)) = ((\bullet), (\bullet), (\bullet))$$

$$K_{a_\gamma}((\bullet, \ldots), (\bullet)) = ((\bullet), (\bullet), (\bullet))$$

$$K_{a_\gamma}((\bullet), (\bullet, \ldots)) = ((\bullet), (\bullet), (\bullet))$$

Therefore, by proving semantic equivalence, the correctness of the clustering algorithms will also be proven. The opposite, that a correct (in the sense of always producing maximum output from minimal input) clustering algorithm will produce a semantically equivalent $K_{a_\gamma}$ from $K_{g_\gamma}$ is trivially true.

Before tackling the correctness proof for the clustering algorithm, some preparation is required. First, the semantic equivalence proof requires that for each
pair of reachable states $q_x, q_y \in Q$ with $q_y \geq q_x$, a sequence of statically scheduled actor firings exists that can be executed in the cluster state $q_x$ resulting in the destination state $q_y$. The proof for the existence of $a$ is derived from the max state reachability lemma given below.

**Max State Reachability**

The notation $q_x \xrightarrow{a} q_y$ is used to denote the existence of a sequence of statically scheduled actor firings $a$, which can be executed in state $q_x$ leading to state $q_y$. Furthermore, a state $q$ is called reachable if and only if $q_0 \xrightarrow{a} q$, and a state $q_y$ is called reachable from $q_x$ if and only if $q_x \xrightarrow{a} q_y$. In the following, the reachability of state $q_m = \max(q_x, q_y)$ from both states $q_x$ and $q_y$ is proven under the precondition that these two states are reachable themselves.

**Lemma 4.1** (Max State Reachability [FZHT13*]). If two states $q_x, q_y \in Q$ are reachable, the pointwise maximum of these two states $q_m = \max(q_x, q_y)$ is reachable from both $q_x$ and $q_y$.

**Proof.** [FZHT13*] The proof is based on the well-known freedom of conflict property of SDF systems. The property can be derived by considering SDF as a subclass of place/transition Petri nets (p/t-nets, for short), where actors correspond to transitions and channels to places. In general p/t-nets, firing a transition can disable another transition which was previously enabled. This situation is referred to as a conflict. However, for the subclass of p/t-nets corresponding to SDF, this can never happen [Mur89]. Therefore, for SDF systems, an actor $a$, once enabled, cannot be disabled by firing another actor $a' \in A \setminus \{a\}$.

Without loss of generality, only the existence of $q_x \xrightarrow{a_{x,m}} q_m$ is proven. Given two reachable states $q_x (q_0 \xrightarrow{a_x} q_x)$ and $q_y (q_0 \xrightarrow{a_y} q_y)$ as well as define the set $A'$ to be the set of actors which have more occurrences in $a_y$ than in $a_x$, i.e., $A' = \{a \in A \mid \#_a a_y > \#_a a_x\}$, then the proposition $q_x \xrightarrow{a_{x,m}} q_m$ is trivially true if $A'$ is empty since $q_m = q_x$. Otherwise, let $a'_y$ be the longest prefix of $a_y$ with the property that each actor $a \in A'$ occurs less than or equal to $\#_a a_x$. Furthermore, due to $\forall a \in A \setminus A' : \#_a a'_y \leq \#_a a_y \leq \#_a a_x$, all actors $a \in A$ occur less than or equal to $\#_a a_x$ in the sequence of statically scheduled actor firings $a'_y$. Moreover, after executing the sequence $a'_y$, at least one enabled actor $a' \in A'$ exists. Therefore, due to the SDF freedom of conflict property, this actor $a'$ must also be enabled after executing sequence $a_x$. Firing $a'$ in state $q_x$ will lead to a new state $q'_x$ with property $q_m = \max(q'_x, q_y)$. Furthermore, the existence of a

---

The notation $\#_x x$ is used to denote the number of occurrences of a value $x$ in a sequence $x$, e.g., $\#_{a_x}(a_x, a_y, a_z) = 2$ denotes that the actor $a_x$ occurs twice in the sequence of statically scheduled actor firings $a = (a_x, a_y, a_z)$.
Next, the proof of the existence of a sequence of statically scheduled actor firings $q_x \xrightarrow{a} q_y$ for all $q_y \geq q_x$ with $q_x, q_y \in Q_\gamma^\infty$ is sketched using the max state reachability lemma. This lemma requires the reachability of the states $q_x$ and $q_y$. First, note that all states $q \in Q_{\text{ini}}^\infty$ are trivially reachable by construction, i.e., the definition of the input/output dependency function (cf. Definition 4.7 on page 138) implies the existence of a sequence $q_0 \xrightarrow{a} q$ for all states $q \in Q_{\text{ini}}^\infty$. Furthermore, the set of initial input/output dependency states are derived from the set of input/output dependency function values by determining the maximal number (cf. Definition 4.9 on page 140) of output actor firings possible with the input actor firings given by an input/output dependency function value. The remaining states $Q_\gamma^\infty \setminus Q_{\text{ini}}^\infty$ are derived by pairwise maximum operations from $Q_{\text{ini}}^\infty$ via a least fixed-point operation (cf. Definition 4.10 on page 141). Let $q_m = \max(q_x, q_y)$ be a state, which is added by a step of the lfp operation, then $q_0 \xrightarrow{a_x \gamma} q_m$ is a sequence of statically scheduled actor firings that reaches $q_m$.

With the above reasoning, it is proven that all states $q \in Q_\gamma^\infty$ are reachable and that $\forall q_x, q_y \in Q_\gamma^\infty : q_y \geq q_x \implies q_x \xrightarrow{a_x \gamma} q_y$ exists. However, the existence of a sequence $\rho_{x,y}$ of rule applications realizing the sequence of statically scheduled actor firings $a_{x,y}$ also needs to be proven.

**Max State Reachability via Rules**

First, some more notation is required: $q_x \xrightarrow{r} q_y$ denotes the existence of a rule $r$ which can be applied in state $q_x$ leading to a new state $q_y$, while $q_x \xrightarrow{\rho_{x,y}} q_y$ denotes the existence of a sequence $\rho_{x,y}$ of rule applications from state $q_x$ to state $q_y$, i.e., $q_y = q_x^{r_{x,0}} \xrightarrow{\rho_{x,0}} q_{x,1}^{r_{x,1}} \xrightarrow{\rho_{x,1}} \ldots \xrightarrow{\rho_{x,l}} q_{x,l} = q_y$. Furthermore, a state $q$ is called rule reachable if and only if $q_0 \xrightarrow{\rho_x} q$, and a state $q_y$ is called rule reachable from $q_x$ if and only if $q_x \xrightarrow{\rho_{x,y}} q_y$. Moreover, rules $r$ are not of an arbitrary form, but obey the rule property (cf. Definition 4.15 on page 149).

In the following, the rule reachability of $q_m = \max(q_x, q_y)$ from both states $q_x$ and $q_y$ is proven, assuming all rules obey the rule property. First, a special case is proven in Lemma 4.2 for rules $r_x$ and $r_y$ that are enabled in a common state $q$. For this limited case, the sequences consist both of a single rule, i.e., $\rho_{x,m} = q_x^{r_{x,k}} \xrightarrow{r_{x,k}} q_m$ and $\rho_{y,m} = q_y^{r_{y,k}} \xrightarrow{r_{y,k}} q_m$, which execute the sequence $a_{x,m}$ and $a_{y,m}$, respectively. Later on, this result will be extended by Lemma 4.3 to all rule reachable states $q_x, q_y \in Q$. From this result, a proof is sketched that all states $q \in Q$ are rule reachable. More formally, the limited case is given below:
Lemma 4.2 (Single-Step Max State Rule Reachability [FZHT13*]). Given two rules \( r_x, r_y \in R \) that can both be applied in state \( q \), i.e., \( r_x.l \leq q \leq r_x.u \) and \( r_y.l \leq q \leq r_y.u \), then the pointwise maximum \( q_\text{m} = \max(q_x, q_y) \) is also reachable from both states \( q_x \) and \( q_y \) via the two rules \( r^i_x \) and \( r^i_y \) given in Definition 4.17 on page 152, i.e., \( q_x \xrightarrow{r^i_x} q_m \) and \( q_y \xrightarrow{r^i_y} q_m \).

Proof. [FZHT13*] Without loss of generality, \( r_x \) is selected to be applied first. Therefore, rule application of \( r_x \) will lead to the state \( q \xrightarrow{r} q + r_x.\eta = q_x \). Hence, the rule \( r^i_x \) is selected for application.\(^77\) For \( r^i_y \) to execute, its precondition \( r^i_y.l \leq q_x \leq r^i_y.u \) has to be satisfied. First, the lower bound \( r^i_y.l \leq q_x \) is considered. With (Definition 4.17) \( r^i_y.l = r_y.l + \min(r_y.\eta, r_x.\eta) \leq r_y.l + r_x.\eta \leq q + r_x.\eta = q_x \), the lower bound is handled. For the upper bound \( q_x \leq r^i_y.u \), two cases can occur for an actor \( a \): First, \( r_y.\eta(a) \leq r_x.\eta(a) \). In this case, \( r^i_y.\eta(a) = r_y.\eta(a) - \min(r_y.\eta(a), r_x.\eta(a)) = 0 \), and, by construction \( q_x(a) \leq r^i_y.u(a) = \infty \). Otherwise, \( r_y.\eta(a) > r_x.\eta(a) \), and, by construction \( r^i_y.u(a) = r_y.u(a) + \min(r_y.\eta(a), r_x.\eta(a)) = r_y.u(a) + r_x.\eta(a) \geq q(a) + r_x.\eta(a) = q_x(a) \). After proving the precondition, \( r^i_y \) is executed and the resulting state \( q' \) is calculated to be \( q' = q_x + r_y.\eta - \min(r_y.\eta, r_x.\eta) = q + r_x.\eta + r_y.\eta - \min(r_y.\eta, r_x.\eta) = q + \max(q_y.\eta, r_x.\eta) = \max(q, q_y.\eta, q + r_x.\eta) = \max(q_y, q_x) = q_m \). \( \square \)

In the following, the general case with arbitrary \( q_x, q_y \in Q \) is discussed. The lemma only requires that there is a common state \( q_0 \) from which both \( q_x \) and \( q_y \) are rule reachable. Note that this is trivially true for the set of initial input/output dependency states \( q \in Q^\text{ini} \) as the derivation of the set of initial input/output dependency rules in Section 4.4.3 ensures that for each \( q \in Q^\text{ini} \) a sequence of rule applications from \( q_0 \) exists. An equivalent argument to the one given at the end of Section 4.4.5 for the extension of the reachability of all states \( q \in Q \) from the reachability of the set of initial input/output dependency states \( q \in Q^\text{ini} \) can be applied to extend the rule reachability from the set of initial input/output dependency states to all states. However, this argument requires a rule equivalent for Lemma 4.1, which is given below:

Lemma 4.3 (Multi-Step Max State Rule Reachability [FZHT13*]). For every two states \( q_x, q_y \in Q \) that are rule reachable from a common state \( q_0 \), i.e., \( q_0 \xrightarrow{\rho_x} q_x \) and \( q_0 \xrightarrow{\rho_y} q_y \), the state \( q_m = \max(q_x, q_y) \) is rule reachable from both \( q_x \) and \( q_y \) via sequences of rule applications \( \rho_{x,m} \) and \( \rho_{y,m} \), i.e., \( q_x \xrightarrow{\rho_{x,m}} q_m \) and \( q_y \xrightarrow{\rho_{y,m}} q_m \).

Proof. [FZHT13*] The lemma is proven via induction. Given that \( q_x = q_{x,i} \) and \( q_y = q_{y,j} \) (cf. Figure 4.47) are reachable from a state \( q_0 = q_{x,0} = q_{y,0} \) via two
\(^77\)The existence of a sequence of statically scheduled actor firings \( a_{x,m} \) for \( r^i_x \) to execute has been proven by Lemma 4.1.
sequences \( \rho_x (q_{x,0} \xrightarrow{r_{x,1}} q_{x,1} \xrightarrow{r_{x,2}} \ldots \xrightarrow{r_{x,i}} q_{x,i}) \) and \( \rho_y (q_{y,0} \xrightarrow{r_{y,1}} q_{y,1} \xrightarrow{r_{y,2}} \ldots \xrightarrow{r_{y,i}} q_{y,i}) \). The maximum indices \( i \) and \( j \) denote the length, i.e., the number of rule applications, for the sequences \( \rho_x \) and \( \rho_y \), respectively.

First, note that for the case of \( i \) being zero, the lemma is trivially true, i.e., if \( i \) is zero then \( q_x = q_0 \) and \( q_y \geq q_x \), therefore \( q_m = q_y \) which is reachable from \( q_0 = q_x \) via the sequence \( \rho_y \). For \( j \) being zero, the symmetrical argument applies.

Hence, it can be assumed that \( i, j \geq 1 \) and that the lemma is true for sequences

\[
\rho_x^i (q_x^1 = q_{x,1} \xrightarrow{r_{x,2}} q_{x,2} \xrightarrow{r_{x,3}} \ldots \xrightarrow{r_{x,i}} q_{x,i} = q_x^i) \] of length \( i - 1 \) and \( \rho_y^j (q_y^1 = q_{y,1} \xrightarrow{r_{y,2}} q_{y,2} \xrightarrow{r_{y,3}} \ldots \xrightarrow{r_{y,j}} q_{y,j} = q_y^i) \) of length \( j - 1 \). The sequences \( \rho_x^i \) and \( \rho_y^j \) are used as induction hypothesis for the proof.

The induction proceeds by proving the existence of two rules \( q_x \xrightarrow{r_{y,1}} q_x^1 \) and \( q_y \xrightarrow{r_{y,1}} q_y^1 \) as well as two sequences \( \rho_x^i \) and \( \rho_y^j \) such that \( q_m = \max(q_x^i, q_y^j) \). With a successful proof of the existence of \( \rho_x^i, r_y^1, \rho_y^j, r_x^1 \), the induction hypothesis can be applied and, hence, the proof of Lemma 4.3 finished.

First, \( \max(q_{x,1}, q_{y,1}) \) is chosen for \( q_x^1 \). Without loss of generality, only the existence of \( \rho_x^i \) is proven. Note that Lemma 4.2 applies to the rules \( r_{x,1} \) and \( r_{y,1} \), therefore the presence of the rule \( r_{y,1}^1 \) is ensured. Now, assuming the presence of the rule \( q_{x,n-1} \xrightarrow{r_{y,1}^{n-1}} q_{x,n-2} \) note that Lemma 4.2 applies to the rules \( r_{y,1}^{n-1} \) and \( r_{x,n} \) which results in the rules \( r_{y,1}^{n-1} \) and \( r_{x,n}^1 \). Furthermore, \( q_x^1 = \max(q_{y,1}^1, q_{x,1}^1, q_{x,2}^1, \ldots, q_{x,i}^1) = \max(q_{y,1}, q_{x,i}) = \max(q_{y,1}, q_x^i) \). The symmetrical argument applies for \( \rho_y^j \) with \( q_y^1 = \max(q_{x,1}^1, q_y^1) \). Finally, note that \( \max(q_{x,1}, q_{y,1}) = \max(q_x^i, q_y^j) = q_m \).

\[ \blacksquare \]
Semantic Equivalence

In this section, the correctness of the presented clustering algorithms is finally proven by showing the equivalence of the semantics of a static cluster $g_\gamma$ and its corresponding composite actor $a_\gamma$. This is done by defining the semantics of both, $g_\gamma$ and $a_\gamma$, with the denotational semantics of KPN and abstracting from data values. The actual proof is done by induction.

**Theorem 4.5** (Semantic Equivalence [FZHT13*]). Given a static cluster $g_\gamma$ of a DFG $g$ satisfying the clustering condition in Definition 4.4 and its corresponding composite actor $a_\gamma$ constructed by either the $\xi_{FSM}$ or the $\xi_{rules}$ cluster refinement operator. Then, it holds that the behaviors of $g$ and the refined DFG $\tilde{g}$ resulting from replacing $g_\gamma$ with $a_\gamma$ are sequence equivalent.

**Proof.** [FZHT13*] Theorem 4.5 is proven by showing that the Kahn descriptions $\mathcal{K}_{g_\gamma}$ and $\mathcal{K}_{a_\gamma}$ of $g_\gamma$ and $a_\gamma$ are equivalent, i.e., $\mathcal{K}_{g_\gamma} \equiv \mathcal{K}_{a_\gamma}$. These functions map tuples of sequences of tokens on the input ports to tuples of sequences of tokens on the output ports.

However, due to the data independent nature of SDF and CSDF actors, the sequences can be abstracted to their length. The production of the correct token values is guaranteed by the firing of the contained actors on the same tokens as in the original subcluster. Finally, the sequence length produced by an input or output actor can be represented by the number of input or output actor firings. With these abstractions, the functions $\mathcal{K}'_{g_\gamma}, \mathcal{K}'_{a_\gamma} : \mathbb{N}_0^{|A_I|} \rightarrow \mathbb{N}_0^{|A_O|}$ are used, which map the number of input actor firings into the number of output actor firings.

Thus, the equivalence of the denotational Kahn functions is reduced to the equivalence of their corresponding abstracted functions, i.e., $\mathcal{K}_{g_\gamma} \equiv \mathcal{K}_{a_\gamma} \iff \mathcal{K}'_{g_\gamma} \equiv \mathcal{K}'_{a_\gamma}$.

Note that for all inputs $\mathcal{K}'_{a_\gamma}$ can only be less than $\mathcal{K}'_{g_\gamma}$ as Lemma 4.1 ensures the existence of sequences of statically scheduled actor firings for all rules, i.e., the non-equivalence of $\mathcal{K}'_{g_\gamma}$ and $\mathcal{K}'_{a_\gamma}$ could only be caused by missing output actor firings of $\mathcal{K}'_{a_\gamma}$. The proof proceeds by showing a contradiction. Assume that for some $\eta \in \mathbb{N}_0^{|A_I|}$ the composite actor lacks output actor firings, i.e., $\mathcal{K}'_{a_\gamma}(\eta) < \mathcal{K}'_{g_\gamma}(\eta)$, and that no sequence $\rho$ exists adding these missing output actor firings. Let the state $q_{def} = (\eta, \mathcal{K}'_{g_\gamma}(\eta))$ be this defective state and $q_{ok} = (\eta, \mathcal{K}'_{a_\gamma}(\eta))$ the correct state. Then, there exists at least one output actor $a_{out}$ with missing firings, i.e., $q_{def}(a_{out}) < q_{ok}(a_{out})$, and a corresponding initial input/output dependency cluster state $q_{ini} \in Q_{ini}^\infty$ where this output is maximized, i.e., $q_{ini}(a_{out}) = q_{ok}(a_{out})$. Then, by the least fixed-point operation from Definition 4.10, the state $q_{corr} = \max(q_{ini}, q_{def})$ must also exist. Furthermore,

\footnote{The relation between Kahn’s denotational semantics and the semantics of data flow models with the notion of firing is presented in [Lee97].}
4. Clustering

\[ i_1 \rightarrow o_1 \]

\[ a_{env} \]

\[ c_{in} = 4 \]

\[ c_{out} = 3 \]

\[ \text{(a) Unrefined DFG } g_{ex_8} \text{ with its cluster } g_{\gamma_{1,2}} \]

\[ \text{(b) Refined DFG } g_{\text{ex}_8} \text{ with the SDF composite actor } a_{\gamma_{1,2}} \]

Figure 4.48: Depicted above is a DFG \( g_{ex_8} \) containing a dynamic actor \( a_{env} \) (white) and a cluster \( g_{\gamma_{1,2}} \) containing the static actors \( a_1 \) and \( a_2 \) (shaded).

by Lemma 4.3, there exists a sequence \( q_{\text{def}} \xrightarrow{s} q_{\text{corr}} \). If there are still output actor firings missing (from output actors other than \( a_{out} \)), then the previous steps can be repeated until \( q_{ok} \) is reached.

4.4.6 FIFO Channel Capacity Adjustments

In contrast to the data flow models that are introduced in literature, DFGs used for refinement of applications into implementations on multi-core platforms including both hardware and software-programmable resources require FIFO channels with limited capacities. Especially, existing synthesis back-ends might only support the synthesis of FIFOs with fixed capacities that are no longer changeable at run time. But, the most severe problem is that QSSs, in general, reduce the accessible capacities of channels contained in a cluster, e.g., channel \( c_{1 \rightarrow 2} \) contained in the cluster \( g_{\gamma_{1,2}} \) depicted in Figure 4.48. Known QSS, e.g., as introduced in Sections 4.4.2 and 4.4.3 as well as similar methods known from literature [PBL95, TBG+13], might—as will be shown—introduce artificial deadlocks due to these inaccessible capacities. The channel capacities of the FIFO channels contained in the cluster can be easily determined in the scheduling phase (cf. Section 4.4.4) that translates the partial repetition vectors of the transitions or rules into sequences of statically scheduled actor firings. However, the channel capacities of the FIFO channels connected to the cluster input and output ports as specified in the original unrefined DFG may induce artificial deadlocks for the refined DFG.

To exemplify, consider the DFG \( g_{ex_8} \) depicted in Figure 4.48. Each initial token of a channel is depicted as a black dot inside the circle representing the channel. The channel capacities are specified via the \text{size} function. The transformation of the DFG \( g_{ex_8} \) into the DFG \( g_{\text{ex}_8} \) illustrates the refinement of the (static) cluster \( g_{\gamma_{1,2}} \) into the composite actor \( a_{\gamma_{1,2}} \). Moreover, as will be
explained later, to prevent any deadlock, the channel capacities of the FIFO channels $c_{in}$ and $c_{out}$ have to be adjusted from capacities to hold at most 4 and 3 tokens to capacities accommodating 7 and 5 tokens, respectively.

The composite actor $a_{\gamma_{1,2}}$ from Figure 4.48 has a cluster FSM with one state $q_0$ and one self-loop transition $t_1$ that, when taken, calls the action $f_{(a_1,a_2,a_2,a_2)}$. This action executes the sequence $(a_1,a_2,a_2,a_2)$ of statically scheduled actor firings, i.e., fire actor $a_1$ once followed by three consecutive firings of actor $a_2$. Moreover, the guard for the transition $t_1$ is given via the notation $\#i_1 \geq 3 \land \#o_1 \geq 3$ that encodes the requirement for the availability of at least three tokens and at least three free places in the channels $c_{in}$ and $c_{out}$ connected to the input port $i_1$ and the output port $o_1$ of the composite actor $a_{\gamma_{1,2}}$. However, as the capacity of channel $c_{out}$ is limited to three tokens—two of them already occupied—, the transition cannot be taken. As a consequence, the composite actor $a_{\gamma_{1,2}}$ depicted in Figure 4.48b would deadlock without a proper channel capacity adjustment.

In the following, the problem this section is dedicated to will be given more formally. Subsequently, the reasons for the introduction of artificial deadlocks due to limited channel capacities will be detailed and a FIFO channel capacity adjustment algorithm will be given to prevent the introduction of artificial deadlocks caused by these reasons.

**Channel Capacity Adjustment Problem**

Note that the determination of limited channel capacities that do not introduce any deadlock for a KPN is in general undecidable [Par95]. Moreover, the channel capacities for the FIFO channels connected to the cluster input and output ports of a cluster cannot be used unmodified from the original unrefined DFG for the refined DFG containing composite actors that implement a QSS. Thus, the channel capacities of the cluster input and output channels cannot be recomputed from scratch, but must be derived by computing an adjustment for the worst case that enlarges the channel capacities of the cluster input and output channels of the original unrefined DFG, e.g., the channel capacities $\text{size}(c_{in})$ and $\text{size}(c_{out})$ have to be increased in order to make the refinement of the DFG $g_{ex}$ into the DFG $\tilde{g}_{ex}$ via application of a QSS for the actors $a_1$ and $a_2$ feasible. More formally, given a cluster $g_\gamma$ and a QSS $g_\gamma.\mathcal{R}$ for it, the problem is to find a vector $g_\gamma.\text{adj} \in \mathbb{N}_{0}^{\{g_{\gamma.1\cup g_{\gamma.2}}\}}$ such that given any deadlock free DFG $g$ exhibiting KPN semantics and containing the cluster $g_\gamma$, then replacing the cluster with its corresponding composite actor $a_\gamma$ and increasing the channel capacities of the channels connected to this composite actor by the amount given by the $g_\gamma.\text{adj}$ vector results in a refined DFG $\tilde{g}$ that is also free of any deadlock.

For the example shown in Figure 4.48, the adjustment vector computed by the algorithm introduced in the next section is $g_{\gamma_{1,2}}.\text{adj} = (n_{c_{in}}, n_{c_{out}}) = (3, 2)$. 

169
4. Clustering

Channel Capacity Adjustment Algorithm

In the following, the reasons for the introduction of artificial deadlocks due to limited channel capacities will be detailed and our novel FIFO channel capacity adjustment algorithm will be introduced. This adjustment algorithm ensures that cluster refinement will not introduce an artificial deadlock into the refined DFG if the original unrefined DFG is a KPN. Note that the presented algorithm does not require any knowledge—indeed, the key aspect of the problem at hand is exactly the lack of such knowledge—of the cluster environment. Nevertheless, for illustration purposes, cluster environments will be modeled by SDF or CSDF actors in the examples presented in this section.

In general terms, a QSS ensures that there are no artificial deadlocks caused by missing tokens on feedback loops from the cluster output ports to the cluster input ports. If the channel capacities are limited, however, clustering might introduce artificial deadlocks due to back pressure. Back pressure induced deadlock denotes the fact that there exists a (reverse) feedback loop of channels and actors where all actors in the loop cannot fire because free places are missing on their output ports connected to the channels contained in the loop. Five reasons can be identified for the introduction of artificial deadlocks due to limited capacities. These reasons are caused by back pressure from

- (A) atomic token consumption on a sole cluster input port,
- (B) atomic token production on a sole cluster output port,
- (C) missing free places on feedback loops from the cluster input ports to the cluster output ports, i.e., input-to-output back pressure,
- (D) missing free places on feedback loops from one cluster input port to another cluster input port, i.e., input-to-input back pressure, and
- (E) missing free places on feedback loops from one cluster output port to another cluster output port, i.e., output-to-output back pressure.

Indeed, reason (A) is just a special case of reason (D) when the cluster has only one input port and reason (B) is the special case of reason (E) when the cluster has only one output port. That is to say, reason (A) can be thought of missing free places on a feedback loop of the input port with itself. Note that the consumption of tokens does also produce free places and, thus, can form part of a feedback loop with itself. Thus, reason (B) can be thought of missing free places on a feedback loop of the output port with itself. Nonetheless, the detailed explanation of the adjustment algorithm will begin with reason (A) and (B) due to the relative simple nature of the problem. Next, the channel capacity adjustments $adj_{i2o}$ required to solve reason (C) concerned with deadlocks due to
4.4 Cluster Refinement in Dynamic Data Flow Graphs

back pressure from a cluster input port to a cluster output port will be discussed. Subsequently, the solutions given by the channel capacity adjustments $\text{adj}_{i2i}$ and $\text{adj}_{o2o}$ to the general cases of reasons (A) and (B) embodied in the reasons (D) and (E) covering the situations that the cluster has *multiple input and output ports* are discussed.

Note that the cluster state space may contain more than one state and that the *cluster environment* is not aware of the current state of a cluster. In the examples used in this paper, the cluster environment is represented by a single actor named $a_{\text{env}}$. In reality, however, it represents the arbitrary complex subgraph of the DFG $g$ that represents the DFG without its contained cluster. Moreover, a deadlock-free execution of the DFG must be guaranteed for each state. Thus, the following analyses of the required channel capacity adjustments given by the three functions $\text{adj}_{i2o}$, $\text{adj}_{i2i}$ and $\text{adj}_{o2o}$ are performed for each state of the cluster state space. With these three adjustment functions given in Definitions 4.21 to 4.23, the FIFO channel capacity adjustment vector $g_\gamma.\text{adj}$ can be defined as follows:

**Definition 4.20. (FIFO Channel Capacity Adjustment Vector)** The FIFO channel capacity adjustment vector $\text{adj} \in \mathbb{N}_{0}^{I \cup O}$ specifies a sufficient increase in channel capacities for each cluster input and output channel such that given any deadlock free DFG $g$ exhibiting KPN semantics and containing the cluster $g_\gamma$, then replacing the cluster with its corresponding composite actor $a_\gamma$ and increasing the channel capacities of the channels connected to this composite actor by the amount given by the adjustment vector results in a refined DFG $\tilde{g}$ that is also free of any deadlock. In the following, the notation $\max X$ is used to denote the least upper bound of a set of partially ordered vectors, e.g., $\max\{(1,0),(0,1)\} = (1,1)$. With this notation, the adjustment vector is derived from the three adjustment functions $\text{adj}_{i2o}$, $\text{adj}_{i2i}$ and $\text{adj}_{o2o}$ as follows:

\[
\pi_{O}(\text{adj}) = \max\{ \text{adj}_{i2o}(q) \mid q \in Q_\gamma \} \tag{4.44}
\]
\[
n_{i2i} = \max\{ \text{adj}_{i2i}(q) \mid q \in Q_\gamma \} \tag{4.45}
\]
\[
n_{i2o} = \max\{ \text{adj}_{i2o}(q, \pi_{O}(\text{adj})) \mid q \in Q_\gamma \} \tag{4.46}
\]
\[
\pi_{I}(\text{adj}) = \begin{cases} 
\max\{ n_{i2i}, n_{i2o} \} & \text{if } O \neq \emptyset \\
 n_{i2i} & \text{otherwise} 
\end{cases} \tag{4.47}
\]

As shown in Equations (4.44) to (4.46), the channel capacity adjustment vector $g_\gamma.\text{adj}$ is derived from the pointwise maximum of the adjustments required by the individual states, thus assuming a worst case cluster environment with respect to back pressure. First, if the cluster has any output ports, then the FIFO channel capacity adjustment vector $\pi_{O}(\text{adj})$ for the cluster output channels is determined by computing the pointwise maximum (see Equation (4.44)) of the adjustments required for the output-to-output back pressure problem for
4. Clustering

each individual cluster state. Next, if the cluster has any input ports, then a vector $n_{i2i}$ of input channel capacity adjustments is determined by computing the pointwise maximum (see Equation (4.45)) of the adjustments required for the input-to-input back pressure problem for each individual cluster state. Subsequently, if the cluster has both input as well as output ports, then a vector $n_{i2o}$ of input channel capacity adjustments is determined by computing the pointwise maximum (see Equation (4.46)) of the adjustments required for the input-to-output back pressure problem for each individual cluster state. Here, the fact that the cluster output channel capacities will be enlarged by the vector specified by $\pi_O(\text{adj})$ can be used to minimize the required adjustments on the cluster input channels. Otherwise, if the cluster has no output ports, then the vector $n_{i2o}$ is assumed to be the all zeros vector. Finally, if the cluster has any input ports, then the FIFO channel capacity adjustment vector $\pi_I(\text{adj})$ for the cluster input channels is determined by computing the pointwise maximum (see Equation (4.47)) of the adjustment requirements of the input-to-input as well as input-to-output back pressure problems.

To exemplify the computation of the $\text{adj}$ vector for the example given in Figure 4.48, its three constituting parts are in Table 4.3. Later on, it will be shown how to compute these function values. The QSS for cluster $g_{\gamma_{1,2}}$ has only a single state. Hence, inserting the values from Table 4.3 into Definition 4.20 results in the following computation of the FIFO channel capacity adjustment vector $\text{adj}$:

$$
\pi_O(\text{adj}) = \text{adj}_{o2o}(q_0) = (2) \\
n_{i2i} = \text{adj}_{i2i}(q_0) = (0) \\
n_{i2o} = \text{adj}_{i2o}(q_0, \pi_O(\text{adj})) = (3) \\
\pi_I(\text{adj}) = \max\{ n_{i2i}, n_{i2o} \} = (3)
$$

In the following, the ‘$g_\gamma$’-prefix will be dropped from various notations when the cluster is clear from context, e.g., $O$ instead of $g_\gamma_O$.

<table>
<thead>
<tr>
<th>$\text{adj}_{o2o}(q)$</th>
<th>$\text{adj}_{i2i}(q)$</th>
<th>$\text{adj}_{i2o}(q, (2))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q = q_0$</td>
<td>$(n_{cout})$</td>
<td>$(n_{cin})$</td>
</tr>
<tr>
<td>$\max{ \ldots \mid q \in Q_\gamma }$</td>
<td>$(2)$</td>
<td>$(0)$</td>
</tr>
</tbody>
</table>
Using this adjustment vector \((\text{adj} = (n_{cin}, n_{cout}) = (3, 2))\), the adjusted channel capacities for the refined DFG \(\tilde{\text{g}}_{\text{ex}}\) depicted in Figure 4.48b are derived as follows:

\[
\begin{align*}
\tilde{\text{g}}_{\text{ex}} \cdot \text{size}(c_{in}) &= \text{g}_{\text{ex}} \cdot \text{size}(c_{in}) + n_{cin} = 4 + 3 = 7 \\
\tilde{\text{g}}_{\text{ex}} \cdot \text{size}(c_{out}) &= \text{g}_{\text{ex}} \cdot \text{size}(c_{out}) + n_{cout} = 3 + 2 = 5
\end{align*}
\]

After having introduced the algorithm to compute the FIFO channel capacity adjustment vector, it will be discussed how to derive its constituting parts embodied by the three adjustment functions \(\text{adj}_{i2o}, \text{adj}_{i2i}\) and \(\text{adj}_{o2o}\). First, the input-to-input back pressure and output-to-output back pressure problems will be considered in isolation for the special cases that a cluster has only a single input or output port, that is reasons (A) and (B), respectively. Subsequently, a cluster that has both a single input and a single output port will be discussed. Hence, capacity adjustments for this cluster will also require the solution of the deadlock problem due to reason (C) concerned with deadlocks due to back pressure from a cluster input port to a cluster output port. Next, the general cases of the input-to-input back pressure and output-to-output back pressure problems (reasons (D) and (E)) will be considered in isolation by analyzing two examples, one having two inputs and the other one two output ports, respectively. Finally, the computation of the FIFO channel capacity adjustment vector will be demonstrated for the general case resulting from the interplay of all reasons (A) to (E) embodied by an example having multiple cluster states as well as multiple input and output ports.

**Atomic Consumption/Production on a Single Cluster Port**

Reasons (A) and (B) stem from the fact that multiple firings of the same input or output actor may be performed atomically by a transition. As previously defined, an input actor denotes an actor of a cluster that consumes tokens from a channel outside the cluster. Conversely, an output actor denotes an actor of a cluster that produces tokens onto a channel outside the cluster. Consider Figure 4.49 as an example for reason (A). In the unrefined DFG \(g_{\text{ex}}\), the input actor \(a_3\) can fire independently of actor \(a_4\). Hence, the two tokens in the cluster input channel \(c_{in}\) can be consumed by two firings of actor \(a_3\), thus allowing actor \(a_{\text{env}}\) to produce three new tokens onto channel \(c_{in}\). In contrast to this, the composite actor \(a_{\gamma_{3,4}}\) of the refined DFG \(\tilde{g}_{\text{ex}}\) can only consume three tokens atomically. Hence, an artificial deadlock would result without adjustment of the channel capacity of the channel \(c_{in}\) from three to five tokens, i.e., the channel capacity \(\text{size}(c_{in})\) is increased by two tokens. To compute the required increase of the channel capacity of \(c_{in}\), the consumption rate of the transition \(t_1\) of the composite actor \(a_{\gamma_{3,4}}\) is first determined to be three tokens, i.e., \(\text{cons}(t_1) = (n_{cin}) = (3)\). In contrast, an isolated firing of the actor \(a_3\) requires one token on the cluster input channel \(c_{in}\), i.e., \(\text{cons}(c_{in}) = 1\). Hence, the capacity of the
4. Clustering

\[ \text{cons}(c_{in}) = 1 \quad \begin{array}{c} \text{cons}(c_{3 \rightarrow 4}) = 3 \end{array} \]

\[ \text{size}(c_{3 \rightarrow 4}) = 3 \]

\[ g_{\gamma_{3,4}} \]

\[ a_{env} \]

\[ c_{in} \quad \text{size}(c_{in}) = 3 \]

(a) Unrefined DFG \( g_{ex9} \) with its cluster \( g_{\gamma_{3,4}} \)

\[ \text{prod}(c_{out}) = 1 \quad \begin{array}{c} \text{prod}(c_{4 \rightarrow 5}) = 3 \end{array} \]

\[ g_{\gamma_{4,5}} \]

\[ a_{env} \]

\[ c_{out} \quad \text{size}(c_{out}) = 3 \]

(b) Refined DFG \( \tilde{g}_{ex9} \) with the SDF composite actor \( a_{\gamma_{3,4}} \)

\[ \text{prod}(t_1) = (3) \]

\[ \text{prod}(t_1) = (3) \]

\[ q_0 \quad l_1 \]

\[ a_{env} \quad \text{size}(c_{out}) = 5 \]

\[ a_{\gamma_{4,5}} \]

Figure 4.49: Example DFG \( g_{ex9} \) illustrates that the refinement of a cluster into a composite actor without channel capacity adjustment can introduce an artificial deadlock into a DFG even if it is only acting as a sink actor.

\[ \text{cons}(t_1) = (3) \]

\[ \text{size}(c_{3 \rightarrow 4}) = 3 \]

\[ \text{size}(c_{in}) = 5 \]

\[ \text{size}(c_{out}) = 5 \]

\[ \text{size}(c_{out}) = 3 \]

\[ \text{size}(c_{out}) = 5 \]

(a) Unrefined DFG \( g_{ex10} \) with its cluster \( g_{\gamma_{4,5}} \)

(b) Refined DFG \( \tilde{g}_{ex10} \) with the SDF composite actor \( a_{\gamma_{4,5}} \)

Figure 4.50: Example DFG \( g_{ex10} \) illustrates that the refinement of a cluster into a composite actor without channel capacity adjustment can introduce an artificial deadlock into a DFG even if it is only acting as a source actor.

Cluster input channel \( c_{in} \) must be enlarged by two tokens in order to compensate for the difference between the two consumption rates. Reason (B) as shown in Figure 4.50 can be handled analogously for the cluster output channel \( c_{out} \).

More formally, the functions \( \text{adj}_{ii} : \mathbb{Q}_\gamma \rightarrow \mathbb{N}_0^{[i]} \) and \( \text{adj}_{oo} : \mathbb{Q}_\gamma \rightarrow \mathbb{N}_0^{[o]} \) are used to denote the required adjustments of the input respectively output channel capacities due to reason (A) or reason (B). Of course, the value for the function \( \text{adj}_{ii} \) can only be computed if the cluster has at least one input.

174
4.4 Cluster Refinement in Dynamic Data Flow Graphs

port. Conversely, the value for the function $\text{adj}_{o2o}$ can only be computed if the cluster has at least one output port. To exemplify, the adjustment values for the three clusters $g_{\gamma_{1,2}}$, $g_{\gamma_{3,4}}$, and $g_{\gamma_{4,5}}$ depicted in Figures 4.49 to 4.51 are given in Table 4.4. In the simple cases—only one cluster state and only a single input or output port—represented by the clusters $g_{\gamma_{3,4}}$ and $g_{\gamma_{4,5}}$, the FIFO channel capacity adjustment vector can be directly derived from Table 4.4.

In case of cluster $g_{\gamma_{3,4}}$ (see Figure 4.49), the adjustment vector is solely represented by a capacity adjustment for the channel $c_{\text{in}}$ connected to the single cluster input port $i_1$. Next, Equations (4.45) and (4.47) from Definition 4.20 are applied, thus, resulting in Equation (4.48). Finally, the cluster state space of cluster $g_{\gamma_{3,4}}$ has only the single state $q_0$ and, thus, the result is given by Equation (4.49).

\[
\pi_I(g_{\gamma_{3,4}}, \text{adj}) = \max \{ g_{\gamma_{3,4}} \cdot \text{adj}_{i2i}(q) \mid q \in Q_\gamma = \{ q_0 \} \} \quad (4.48)
\]

\[
= g_{\gamma_{3,4}} \cdot \text{adj}_{i2i}(q_0) = (2) \quad (4.49)
\]

Conversely, in case of cluster $g_{\gamma_{4,5}}$ (see Figure 4.50), the adjustment vector is solely represented by an adjustment for the channel capacity for the channel $c_{\text{out}}$ connected to the single cluster output port $o_1$. To proceed, Equation (4.44) from Definition 4.20 is applied to transform the problem into Equation (4.50). To conclude, only the single state $q_0$ is present in the cluster state space, thus, giving the result present in Equation (4.51).

\[
\pi_O(g_{\gamma_{4,5}}, \text{adj}) = \max \{ g_{\gamma_{4,5}} \cdot \text{adj}_{o2o}(q) \mid q \in Q_\gamma = \{ q_0 \} \} \quad (4.50)
\]

\[
= g_{\gamma_{4,5}} \cdot \text{adj}_{o2o}(q_0) = (2) \quad (4.51)
\]

However, if a cluster has both input and output ports, then the input-to-output back pressure problem must also be solved as discussed next.

<table>
<thead>
<tr>
<th>Table 4.4: Adjustment values due to reasons (A) and (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{adj}_{i2i}(q_0)$</td>
</tr>
<tr>
<td>$g_{\gamma_{3,4}}$ (see Figure 4.49)</td>
</tr>
<tr>
<td>$g_{\gamma_{4,5}}$ (see Figure 4.50)</td>
</tr>
<tr>
<td>$g_{\gamma_{1,2}}$ (see Figure 4.51)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{adj}_{o2o}(q_0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_{\gamma_{3,4}}$ (see Figure 4.49)</td>
</tr>
<tr>
<td>$g_{\gamma_{4,5}}$ (see Figure 4.50)</td>
</tr>
<tr>
<td>$g_{\gamma_{1,2}}$ (see Figure 4.51)</td>
</tr>
</tbody>
</table>
4. Clustering

\[
\text{prod}(c_{\text{out}}) = 1
\]

\[
\text{cons}(c_{\text{in}}) = 3
\]

\[
\text{size}(c_{\text{in}}) = 4 \quad \text{size}(c_{\text{out}}) = 3
\]

(a) Unrefined DFG \(g_{\text{ex}8}\) with its cluster \(g_{\gamma 1,2}\)

\[
\text{prod}(t_1) = (3)
\]

\[
\text{cons}(t_1) = (3)
\]

\[
\text{size}(c_{\text{in}}) = 7 \quad \text{size}(c_{\text{out}}) = 5
\]

(b) Refined DFG \(\widetilde{g}_{\text{ex}8}\) with the SDF composite actor \(a_{\gamma 1,2}\)

Figure 4.51: Shown above is the replication of the DFG \(g_{\text{ex}8}\) from Figure 4.48 where the cluster environment \(a_{\text{env}}\) is given by a CSDF actor.

Input-to-Output Back Pressure

While reasons (A) and (B) can occur even if the composite actor is only a sink or source actor, reason (C) can only occur if the cluster has both inputs and outputs. To exemplify, the cluster \(g_{\gamma 1,2}\) shown in Figure 4.48 and replicated here for the convenience of the reader as Figure 4.51 is again considered. The DFG \(g_{\text{ex}8}\) containing this cluster has the following FIFO channel capacities:

\[
\text{size}(c_{\text{in}}) = 4
\]
\[
\text{size}(c_{\text{out}}) = 3
\]
\[
\text{size}(c_{1 \rightarrow 2}) = 4
\]

In the following, a scenario that leads to deadlock due to reasons (B) and (C) will be presented. In this scenario, two firings of the cluster environment \(a_{\text{env}}\) will consume one token from the cluster output channel \(c_{\text{out}}\) and push six tokens onto the cluster input channel \(c_{\text{in}}\) by twice producing three tokens onto the channel. This scenario is also performed by the first two firings of the CSDF actor \(a_{\text{env}}\) that is depicted in Figure 4.51.

Obviously, when the unrefined DFG \(g_{\text{ex}8}\) is dynamically scheduled, the specified FIFO channel capacities are sufficient: Fire actor \(a_1\) once, wait for the first firing of the cluster environment modeled by \(a_{\text{env}}\) to result in two free places on the cluster output channel \(c_{\text{out}}\) and further three tokens on the cluster input channel \(c_{\text{in}}\), fire actor \(a_2\) twice and actor \(a_1\) once, wait for the second firing of the environment to produce three more tokens on the cluster input channel \(c_{\text{in}}\). After this scenario has finished, all channels \(c_{\text{in}}, c_{\text{out}}, c_{1 \rightarrow 2}\) are filled to capacity.

On the other hand, after \(g_{\gamma 1,2}\) has been refined into the composite actor \(a_{\gamma 1,2}\) (see Figure 4.51b), the action \(f_{(a_1,a_2,a_2,a_2)}\) of transition \(t_1\) requires three tokens on
the cluster input port $i_1$ and three free places on the cluster output port $o_1$ before it can be executed. However, assuming that the channel capacity of channel $c_{out}$ is still three tokens, then the transition cannot be enabled due to the two initial tokens in channel $c_{out}$. Hence, no tokens are consumed from the cluster input channel $c_{in}$ and, thus, the cluster environment cannot even produce the first batch of three tokens onto the cluster input channel $c_{in}$. Therefore, the refined graph $\tilde{g}_{ex}$ results in a deadlock. This artificial deadlock is caused by reason (B).

To compensate, the channel capacities must be adjusted (see Table 4.4 cluster $g_{\gamma_{1,2}}$) as follows:

$$\text{size}'(c_{in}) = \text{size}(c_{in}) + \text{adj}_{i2i}(q_0) = 4 + 0 = 4$$

$$\text{size}'(c_{out}) = \text{size}(c_{out}) + \text{adj}_{o2o}(q_0) = 3 + 2 = 5$$

As can be seen, the channel capacity of the cluster output channel $c_{out}$ has been increased by two tokens ($\text{adj}_{o2o}(q_0) = 2$) to handle reason (B). This increase stems from the difference of the atomic production of three tokens by the transition $t_1$ (see Figure 4.51b) as compared to the production of one token by a single firing of actor $a_2$. In contrast to this, reason (A) does not pose a problem ($\text{adj}_{i2i}(q_0) = 0$) for this example. However, these adjustments will prove to be still insufficient to prevent any artificial deadlock due to reason (C).

After the adjustment given above, the composite actor can fire once, thus allowing the cluster environment to produce the first batch of three tokens out of the six tokens. Subsequently, the cluster environment $a_{env}$ will fire once resulting in a state where the cluster input and output channels are both storing four tokens. Hence, neither can the composite actor fire a second time, nor can the cluster environment produce the second batch of three tokens, thus resulting in another artificial deadlock. This deadlock is caused by the inability of the cluster environment to access the FIFO channel capacity of four tokens available on the FIFO channel $c_{1\rightarrow2}$ contained in the composite actor $a_{\gamma_{1,2}}$. That is, the artificial deadlock is due to reason (C).

To compensate, the capacity of either the cluster input channel or the cluster output channel has to be increased. In the following, the function $\text{adj}_{i2o}(q, n)$ will be used to denote a sufficient increase in the channel capacities of the cluster input channels in order to facilitate the compensation for the inaccessible channel capacities inside the cluster. This compensation depends on the given cluster state $q$ and the adjustment $n \in \mathbb{N}_0^{O}$ that will be used to increase the channel capacities of the cluster output channels.

To compute $\text{adj}_{i2o}$, a cluster state $q_{dyn} \in Q_{\gamma}^{dyn}$ that could have been encountered during dynamic scheduling of the cluster is searched for. This cluster state $q_{dyn}$ must have the property that it stores a local maximum of tokens inside the cluster. The number of tokens stored inside a cluster is maximized by starting
from the cluster state \( q \in Q_\gamma \) consuming tokens from the cluster inputs and undoing\(^{80}\) the production of produced tokens on the cluster output ports.

In case of cluster \( g_{1,2} \): the state where a local—even a global—maximum of tokens is stored inside the cluster is given when the channel \( c_{1 \rightarrow 2} \) is filled to capacity. As can be seen in Figure 4.52, two dynamic cluster states \( q_{\text{dyn},1} \) and \( q_{\text{dyn},2} \) of maximum token storage can be derived from the original state \( q_0 \).

The first dynamic cluster state \( q_{\text{dyn},1} \in Q_\gamma^{\text{dyn}} \) (see Figure 4.52b) is reached from the original state \( q_0 \) by firing actor \( a_1 \) once and undoing one firing of actor \( a_2 \), thus consuming three tokens from the input port \( i_1 \) and consuming one token from the output port \( o_1 \). Using the input-to-output back pressure adjustment function \( \text{adj}_{i2o} \), this can also be expressed as \( \text{adj}_{i2o}(q_0, 1) = (3) \) and should be read as: Given an increase of the capacity of \( c_{\text{out}} \) by at least one token and the case that the cluster is in state \( q_0 \), then an increase of three tokens for the capacity of the channel \( c_{\text{in}} \) is sufficient to solve reason (C) concerned with artificial deadlock due to input-to-output back pressure. However, there is still reason (B) that requires an increase of at least two tokens of the capacity of the channel \( c_{\text{out}} \). Hence, the question arises how many additional free places on the cluster input channel are required to solve the input-to-output back pressure problem if the capacity on the cluster output channel has already been increased by two free places, i.e., \( \text{adj}_{i2o}(q_0, 2) \). Still, three more free places are required for the channel \( c_{\text{in}} \) to solve the input-to-output back pressure problem, i.e., \( \text{adj}_{i2o}(q_0, 2) = (3) \), since undoing two firings of actor \( a_2 \) does not fill channel \( c_{1 \rightarrow 2} \) to capacity. Thus, with the function values for the the cluster

\(^{80}\) In reality, the cluster will not undo any actor firing and this should be thought of as delayed production of tokens by a dynamically scheduled cluster in contrast to a composite actor implementing a QSS. A QSS is required to always produce the maximal number of output tokens from a minimal number of input tokens and, hence, will never delay the production of tokens on its output ports.
4.4 Cluster Refinement in Dynamic Data Flow Graphs

In Table 4.4 the values in Table 4.3 are explained, and the updated channel capacities given below can be computed as given in Definition 4.20.

\[
\begin{align*}
g_{\text{ex}}, \text{size}(c_{\text{in}}) &= 4 + 3 = 7 \\
g_{\text{ex}}, \text{size}(c_{\text{out}}) &= 3 + 2 = 5
\end{align*}
\]

The second dynamic cluster state \( q_{\text{dyn}, 2} \in Q_{\gamma}^{\text{dyn}} \) (see Figure 4.52c) is reached from the original state \( q_0 \) by undoing four firings of actor \( a_2 \), thus consuming four token from the output port \( o_1 \), i.e., \( \text{adj}_{a_2}(q_0, 4) = 0 \). Hence, instead of increasing the channel capacity of the cluster input channel \( c_{\text{in}} \), it would also be possible to increase the channel capacity of the cluster output channel \( c_{\text{out}} \) by four tokens to handle reason (C). Thus, leading to the following channel capacities of the cluster input and output channels:

\[
\begin{align*}
g_{\text{ex}}, \text{size}(c_{\text{in}}) &= 4 + 0 = 4 \\
g_{\text{ex}}, \text{size}(c_{\text{out}}) &= 3 + 4 = 7
\end{align*}
\]

More formally, the function \( \text{adj}_{a_2o} \) can be computed as follows:

**Definition 4.21.** (Input-to-Output Back Pressure Adjustment) The input-to-output back pressure adjustment is encoded by the vector-valued function \( \text{adj}_{a_2o} : Q_{\gamma} \times N_0^{|O|} \rightarrow N_0^{|I|} \). Given a cluster state \( q \) and a vector \( n \in N_0^{|O|} \) that denotes an increase in the output channel capacities of the cluster, then this function computes the required increase of the cluster input channel capacities in order to solve the input-to-output back pressure problem for the given state. The adjustment is computed for each input port \( i \) in isolation. A dynamic cluster state \( q_{\text{dyn}} \) (see Equation (4.52)) that could have been encountered during dynamic scheduling of the cluster is determined. In general terms, the dynamic cluster state \( q_{\text{dyn}} \) is encountered during dynamic scheduling when the cluster has already consumed some input tokens, but the production of output tokens has been delayed. Thus, compared to the cluster state \( q \) of the composite actor \( a_{\gamma} \), the dynamic cluster state consumes \( n_i \) more tokens (see Equation (4.53)) from the cluster input port \( i \) but produces \( n_O \) less tokens (see Equation (4.54)) on the cluster output ports \( O \). However, the generated composite actor \( a_{\gamma} \) will always produce the maximal number of output tokens, thus always producing all tokens remaining in the cluster to the cluster output channels. Hence, only dynamic cluster states are considered where these productions of output tokens by the composite actor can be accommodated (see Equation (4.54)) by the increase in the channel capacities of the cluster output channels denoted by the vector \( n \). Moreover, only dynamic cluster states \( q_{\text{dyn}} \in Q_{\gamma}^{\text{dyn}} \) (see Equation (4.55)) that exhibit a local maximum of tokens stored inside the cluster are considered. That is to say, the number of tokens stored inside the cluster cannot be increased by consuming more tokens from the cluster input port \( i \) (see Equation (4.56)) or

\[
\begin{align*}
g_{\text{ex}}, \text{size}(c_{\text{in}}) &= 4 + 0 = 4 \\
g_{\text{ex}}, \text{size}(c_{\text{out}}) &= 3 + 4 = 7
\end{align*}
\]
4. Clustering

![Diagram of DFGs showing original and refined structures](image)

(a) Original unrefined DFG $g_{ex11}$ with its cluster $g_{1,3,4}$

(b) Refined DFG $g_{ex11}$ containing the composite actor $a_{1,3,4}$

**Figure 4.53:** Here, the DFG $g_{ex11}$ is used to generalize the input-to-input back pressure problem to multiple input ports.

Delaying the production of further output tokens (see Equation (4.57)). Finally, to determine the required channel capacity adjustment for the cluster input channel connected to the cluster input port $i$, the dynamic cluster state $q_{dyn}$ is chosen where $n_i$ is minimal (see Equation (4.52)). This is the state where the already present channel capacity increases $n$ on the cluster output channels are best taken advantage of.

\[
\pi_{\{i\}}(\text{adj}_{20}(q, n)) = \min\{n_i \mid \exists q_{dyn} \in Q_{dyn}^{:} : \]
\[
n_i = -\pi_{\{i\}}(\text{io}(q_{dyn}) - \text{io}(q)) \geq 0 \quad \wedge \quad \text{(4.53)}
\]
\[
n_O = -\pi_O(\text{io}(q_{dyn}) - \text{io}(q)) \leq n \quad \wedge \quad \text{(4.54)}
\]
\[
\#q_{dyn} \in Q_{dy}^{:} \setminus \{q_{dyn}\} : \]
\[
n_i \leq -\pi_{\{i\}}(\text{io}(q_{dyn})' - \text{io}(q)) \quad \wedge \quad \text{(4.55)}
\]
\[
n_O \leq -\pi_O(\text{io}(q_{dyn})' - \text{io}(q)) \quad \text{(4.56)}
\]

Input-to-Input Back Pressure

Previously, the input-to-input and output-to-output back pressure problems have only been discussed for the special cases that the cluster has only a single input or a single output port, respectively. In contrast to this, the input-to-input back pressure problem if a cluster has multiple inputs will be discussed in this section. To put focus on this, the DFG $g_{ex11}$ shown in Figure 4.53 that exhibits the input-to-input back pressure problem in isolation will be used to analyze the problem. That is to say, input-to-output and output-to-output back pressure do not pose a problem for this example as the cluster $g_{1,3,4}$ has no output ports.

In the following, a critical scenario that leads to an artificial deadlock due to reason (D) will be presented. As can be seen in Figure 4.53a, the cluster...
environment $a_{\text{env}}$ tries to produce six tokens onto the cluster input channel $c_{\text{in},2}$ by producing two batches of three tokens each. In the unrefined case, there are sufficient channel capacities inside the cluster to facilitate the consumption of six tokens from the cluster input channel $c_{\text{in},2}$, i.e., fire actor $a_1$ once, consuming the first batch of three tokens from cluster input channel $c_{\text{in},2}$, fire actor $a_3$ once, consuming one of the initial tokens provided in the cluster input channel $c_{\text{in},1}$, and fire actor $a_1$ once more, consuming the second batch of three tokens from cluster input channel $c_{\text{in},2}$.

On the other hand, after $g_{\gamma_1,3,4}$ has been refined into the composite actor $a_{\gamma_1,3,4}$ (see Figure 4.53b), the transition $t_1$ requires three tokens on the cluster input ports $i_1$ and $i_2$ before it can be taken. Moreover, as the cluster environment $a_{\text{env}}$ does not provide any additional tokens on the cluster input channel $c_{\text{in},1}$, the transition $t_1$ cannot be taken. Hence, assuming that the channel capacity of the cluster input channel $c_{\text{in},2}$ is still four tokens, an artificial deadlock results due to reason (D). To compensate, the channel capacities must be adjusted as follows:

\[
\begin{align*}
\widehat{g}_{\text{ex}} & \cdot \text{size}(c_{\text{in},1}) = g_{\text{ex}} & \cdot \text{size}(c_{\text{in},1}) + n_{c_{\text{in},1}} = 3 + 2 = 5 \\
\widehat{g}_{\text{ex}} & \cdot \text{size}(c_{\text{in},2}) = g_{\text{ex}} & \cdot \text{size}(c_{\text{in},2}) + n_{c_{\text{in},2}} = 4 + 6 = 10
\end{align*}
\]

These adjustments correspond to the number of tokens that can be consumed by the cluster on its two different input ports while still consuming less tokens on both input ports than would be necessary to activate the transition $t_1$. For a better depiction of the number of consumed tokens, the situation as shown in Figure 4.54a is assumed as start point. As can be seen in Figure 4.54c, the cluster $g_{\gamma_1,3,4}$ can consume up to six tokens on its input port $i_2$ (fire actor $a_3$ once and actor $a_1$ twice) without enabling the transition $t_1$. Thus, the channel capacity of the cluster input channel $c_{\text{in},2}$ must be increased by six tokens, i.e., $n_{c_{\text{in},2}} = 6$. Consumption of more than six tokens would require nine tokens on input port $i_2$ and four tokens on input port $i_1$. Hence, enabling the input/output guard $k^{\text{io}} = \#i_1 \geq 3 \land \#i_2 \geq 3$ of the transition $t_1$. An equivalent observation (see Figure 4.54b) can be made for the input port $i_1$, which can consume up to two tokens without enabling the transition $t_1$. More formally, the function $\text{adj}_{i_2}$ is used to encode the adjustment requirements:

**Definition 4.22.** (Input-to-Input Back Pressure Adjustment) The input-to-input back pressure adjustment is given via the vector-valued function $\text{adj}_{i_2} : \mathbb{Q}_\gamma \rightarrow \mathbb{N}^{|I|}$ that associates with a cluster state $q$ the required increase of the cluster input channel capacities in order to solve the input-to-input back pressure problem for the given cluster state. The adjustment is computed for each input port $i$ by determining the maximal (see Equation (4.58)) number $\pi_{(i)}(n_I)$ of tokens that can be consumed (see Equation (4.59)) by the cluster on the input port $i$, but without enabling (see Equation (4.61)) any transition (see Equation (4.60)) leaving the state $q$. Here, the vector $n_I$ represents the number of
4. Clustering

![Diagram showing three scenarios: (a) Original situation in cluster state $q_0$, (b) Dynamic cluster state $q_{dyn,1}$, and (c) Dynamic cluster state $q_{dyn,2}$](image)

**Figure 4.54:** In the above given example, a cluster environment $g_{env}$ that provides an infinite number of tokens is assumed. For depiction purpose, seven tokens are chosen for each cluster input port. Three situations distinguished by their corresponding state of the cluster are shown. The original situation ($q_0$) is given in (a). The situation ($q_{dyn,1}$) where the cluster consumes the maximal number of tokens (here two tokens) from input port $i_1$ while still not activating transition $t_1$ is given in (b). Conversely, the situation ($q_{dyn,2}$) where the cluster consumes the maximal number of tokens (here six tokens) from input port $i_2$ while still not activating transition $t_1$ is shown (c).

The number of tokens that would be consumed on the cluster input ports during a sequence of actor firings transitioning the cluster from the current cluster state $q$ to another state $q_{dyn} \in Q_{\gamma}^{dyn}$ that could be encountered by dynamic scheduling of the cluster.

\[
\pi_{\{i\}}(adj_{\{i\}}(q)) = \max \{ \pi_{\{i\}}(n_I) \mid \exists q_{dyn} \in Q_{\gamma}^{dyn} : (4.58)
\]

\[
n_I = - \pi_I(io(q_{dyn}) - io(q)) \geq 0 \quad \land \quad (4.59)
\]

\[
\#t \in T \land t.q_{src} = q : (4.60)
\]

\[
n_I \geq \text{cons}(t) \} \quad (4.61)
\]

**Output-to-Output Back Pressure**

Finally, to conclude the discussion of the reasons of artificial deadlock due to back pressure, the solution for output-to-output back pressure will be generalized to clusters having more than one output port. The problem of output-to-output back pressure can even occur if the cluster is a pure source for the DFG. We will use this to consider the problem in isolation by analyzing the output-to-output back pressure problem for the cluster $g_{\gamma_{2,4,5}}$ shown in Figure 4.55. As
4.4 Cluster Refinement in Dynamic Data Flow Graphs

Figure 4.55: For illustration of the generalized solution of the output-to-output pack pressure problem from one to multiple output ports the above given DFG $g_{ex12}$ is employed.

this cluster has no input ports, input-to-output and input-to-input back pressure does not occur.

As always, the description of a critical scenario for the output-to-output back pressure problem in the DFG $g_{ex12}$ is given. In this scenario, the cluster environment $a_{env}$ pulls six tokens from the cluster output channel $c_{out,1}$ by twice consuming three tokens from the channel. This can be accommodated by the unrefined DFG $g_{ex12}$ by firing both actors $a_5$ and $a_2$ once, a consumption of the first batch of three tokens by the cluster environment $a_{env}$, three more firings of actor $a_5$, and the consumption of the second batch of three tokens by the cluster environment $a_{env}$. After this sequence of actor firings, the channel $c_{5\rightarrow2}$ is filled to capacity.

On the other hand, the composite actor $a_{\gamma_{2,4,5}}$ is always producing the maximum number of output tokens, thus always producing all tokens remaining in channel $c_{5\rightarrow2}$ to the cluster output channel $c_{out,2}$. Hence, assuming that the channel capacity of the cluster output channel $c_{out,2}$ is still three tokens, an artificial deadlock results due to reason (E). To compensate, the channel capacities must be adjusted as follows:

$$g_{ex12}.\text{size}(c_{out,1}) = g_{ex12}.\text{size}(c_{out,1}) + n_{c_{out,1}} = 3 + 2 = 5$$
$$g_{ex12}.\text{size}(c_{out,2}) = g_{ex12}.\text{size}(c_{out,2}) + n_{c_{out,2}} = 3 + 5 = 8$$

To compute these adjustment values for the transition $t_1$ and each output port, the maximal number of token productions that can be undone on this port while still not having undone all produced tokens of transition $t_1$ will be determined. To enable a graphical illustration of undoing token productions, the original situation given in Figure 4.56a will be assumed. In this situation, the cluster environment has consumed seven tokens from each cluster output port.
4. Clustering

Figure 4.56: To illustrate the undoing of actor firings of the cluster $g_{2,4,5}$, the situation in (a) is assumed. Here, the cluster environment $g_{env}$ has consumed seven tokens provided on each cluster output port. Moreover, two situations resulting from undoing actor firings starting from (a) are shown in (b) and (c). These situations correspond to the dynamic cluster states $q_{dyn,1}$ and $q_{dyn,2}$, respectively. In particular, state $q_{dyn,1}$ corresponds to the situation where the cluster has undone the maximal number of token productions (here two tokens) on the output port $o_1$ while still not having undone all produced tokens of transition $t_1$. Likewise, state $q_{dyn,2}$ has undone the maximal number of token productions (here five tokens) on the output port $o_2$.

As can be seen in Figure 4.56c, five token productions can be undone on the cluster output port $o_2$ while still allowing the production of at least one token that would also have been produced by the transition $t_1$, e.g., the token produced by the first firing of actor $a_5$ of the transition $t_1$. Hence, the channel capacity of the cluster output channel $c_{out,2}$ has been increased by five tokens, i.e., $n_{c_{out,2}} = 5$. An equivalent observation (see Figure 4.56b) can be made for the output port $o_1$. In this case, two firings of the actor $a_5$ can be undone while still producing at least one token, i.e., the token produced by the first firing of actor $a_5$. Thus, the channel capacity of the cluster output channel $c_{out,1}$ has been increased by two tokens, i.e., $n_{c_{out,1}} = 2$. If the first firing of actor $a_5$ is also undone, then the three firings of actor $a_2$ have also to be reversed. Hence, all of the tokens produced by the transition $t_1$ have been undone. More formally, the function $adj_{o_2o}$ is used to encode the adjustment requirements:

**Definition 4.23. (Output-to-Output Back Pressure Adjustment)** The solution for the output-to-output back pressure problem for a given cluster state is computed by the vector-valued output-to-output back pressure adjustment function $adj_{o_2o} : Q_\gamma \rightarrow \mathbb{N}_0^{|O|}$ that associates with the given cluster state $q$ the required

184
increase of the cluster output channel capacities. The adjustment is computed for each output port \( o \) by determining the maximal (see Equation (4.62)) number \( \pi_{\{ o \}}(n_O) \) of token productions that can be undone (see Equation (4.64)) on an output port \( o \) of the cluster starting from the destination state of a transition \( t \) (see Equation (4.63)) such that dynamic scheduling of the cluster would still be able to produce at least one token (see Equation (4.65)) that would have been produced by the transition \( t \). Here, the vector \( n_O \) represents the number of token productions that could be undone starting from the destination state \( t.q_{\text{dst}} \) of the transition \( t \) such that at least one token produced by the transition could still be produced via dynamic scheduling of the cluster.

\[
\pi_{\{ o \}}(\text{adj}_{o\rightarrow o}(q)) = \max\{ \pi_{\{ o \}}(n_O) \mid \exists q_{\text{dyn}} \in Q_{\gamma}^\text{dyn} : \exists t \in T \land t.q_{\text{src}} = q : \\
(n_O = -\pi_O(\text{io}(q_{\text{dyn}}) - \text{io}(t.q_{\text{dst}})) \geq 0 \land \\
(n_O \not\geq \text{prod}(t)) \}
\]

(4.62)

(4.63)

(4.64)

(4.65)

**General Example Requiring Solutions for All Reasons**

Finally, as an example requiring all three adjustment parts, consider the DFG \( g_{\gamma_1,2,...,6} \) depicted in Figure 4.57. The adjustment requirements to handle the output-to-output, input-to-input, and input-to-output and back pressure problems are listed in Table 4.5. First, the output-to-output back pressure problem is handled. According to Table 4.5, an increase of at least eight tokens is required for the cluster output channel \( c_{\text{out},1} \) and an increase of at least four tokens for the cluster output channel \( c_{\text{out},2} \), i.e., \( \pi_O(\text{adj}) = (8, 4) \). These adjustments are reused for the input-to-output back pressure problem, i.e., \( \text{adj}_{i\rightarrow o}(q, (8, 4)) \). Finally, the required channel capacity adjustments for the cluster input channels are determined from the values of the \( \text{adj}_{i\rightarrow i}(q) \) and \( \text{adj}_{i\rightarrow o}(q, (8, 4)) \) functions. Here, an increase of at least seven tokens is required for the cluster input channel \( c_{\text{in},1} \) and at least six tokens for the cluster input channel \( c_{\text{in},2} \), i.e., \( \pi_I(\text{adj}) = \max\{ (3, 3), (7, 6) \} = (7, 6) \).

**Table 4.5:** The three channel capacity adjustments parts for the cluster \( g_{\gamma_1,2,...,6} \)

<table>
<thead>
<tr>
<th>( q )</th>
<th>( \text{adj}_{o\rightarrow o}(q) )</th>
<th>( \text{adj}_{i\rightarrow i}(q) )</th>
<th>( \text{adj}_{i\rightarrow o}(q, (8, 4)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q = q_0 )</td>
<td>( (2, 4) )</td>
<td>( (2, 3) )</td>
<td>( (7, 0) )</td>
</tr>
<tr>
<td>( q = q_1 )</td>
<td>( (5, 2) )</td>
<td>( (2, 0) )</td>
<td>( (4, 3) )</td>
</tr>
<tr>
<td>( q = q_2 )</td>
<td>( (8, 2) )</td>
<td>( (3, 0) )</td>
<td>( (1, 6) )</td>
</tr>
<tr>
<td>( \max{ \ldots \mid q \in Q_{\gamma} } )</td>
<td>( (8, 4) )</td>
<td>( (3, 3) )</td>
<td>( (7, 6) )</td>
</tr>
</tbody>
</table>
4. Clustering

In this section, the benefits of the proposed clustering methodology are presented. First, in Section 4.5.1, it is shown that clustering can obtain a speedup of 10x and beyond by reducing the overall overhead in the execution of fine-grained data flow graphs that are dominated by static actors. Synthetic and real-word examples are used to illustrate this advantage. Then, in Section 4.5.2, it is shown that the proposed clustering methodology enables a much larger design space for cluster refinements as compared to monolithic clusterings, which are representable by SDF composite actors. Thus, the second key contribution of this work can exploit the presence of static actors in fine-grained data flow graphs in a larger set of situations than would be possible with the traditional monolithic clustering approaches known from literature.

Figure 4.57: As last example, the DFG $a_{\gamma_1,2 \ldots 6}$ having multiple input and output ports as well as a corresponding QSS with multiple states is considered to demonstrate the interplay of all reasons (A) to (E).
4.5 Results

4.5.1 Speedup from Overhead Reduction

In the following, results from [FZHT13*] are presented. Binary executables have been generated by (M1) fully dynamically scheduling the graphs (for reference purpose), (M2) the rule-based approach from Section 4.4.3, and (M3) the automata-based approach from Section 4.4.2. The measurements for the results where conducted on an Intel® Core™ i7-2600 CPU with 3.40GHz. All executables where generated with the -Os compile option using version 4.5 of the gcc compiler suite. For code size measurements, the generated executable have been stripped of debugging information. Furthermore, the binary code sizes produced by synthesizing the clustering with the rule-based approach (M2) and the automata-based approach (M3) were also required to not exceed 133% of the code size of the dynamic case (M1). The generated binary executables are used for speedup and code size measurements to evaluate the approaches against each other.

First, the dynamic scheduling algorithm is presented which is used as a baseline for all comparisons. This scheduling algorithm is a variant of the simple round-robin scheduler with the modification that (1) each actor is fired in a loop until the number of tokens on its inputs are insufficient for further firings and (2) all static actors in a cluster $g_\gamma$ are scheduled by a round-robin subscheduler which is executed by the main round-robin scheduler (cf. Lines 17 to 26 from Algorithm 3).

Modification (2) enables the replacement of the round-robin subscheduler with the automata-based approach or the rule-based approach without any modifications to the check and execute sequence of the dynamic actors by the main round-robin scheduler. This is important as permutations in the check and execute sequence can lead to noticeable performance fluctuations unrelated to the QSS under consideration. The replacement subscheduler to execute the rules derived in Section 4.4.3 is given in Algorithm 4. This subscheduler is used to replace Lines 17 to 26 from Algorithm 3. The resulting scheduler is used to evaluate the performance and code size of the rule-based static data flow clustering algorithm (M2) presented in Section 4.4.3. For evaluation of the automata-based approach (M3), Lines 17 to 26 from Algorithm 3 are replaced with the appropriate subscheduler to execute the cluster FSM derived via the algorithm given in Section 4.4.2.

In order to illustrate the benefits of the clustering algorithm developed in Section 4.4.2, it is applied to both real-world applications (cf. Figures 4.58 and 4.59) as well as synthetic data flow graphs. Shaded vertices correspond to static actors. White vertices are dynamic actors. The first real-world application is an MP3 decoder (cf. Figure 4.58) working on the MP3 granule level (i.e., 576 frequency/time domain samples). The MP3 decoder has five static subclusters (two trivial subclusters consist of a single actor only and one subclusters only
4. Clustering

containing two actors). The remaining two subclusters $g_{\gamma_a}$ and $g_{\gamma_b}$ are processed by the presented automata-based clustering algorithm to compute their QSSs. For these two subclusters, QSSs have been computed using the presented approach. To evaluate the scheduling overhead reduction in isolation, as much functionality as possible is removed from all actors. The dynamic scheduler for this MP3 decoder has an overall execution time of about 184 ms for a given MP3 input stream (approximately 20 MB). This decreases to 57 ms when using the previously computed QSSs for the two subgraphs, i.e., an improvement of approximately 69%. For a real-world test, the unmodified MP3 decoder is used.

Algorithm 3: Fully dynamic scheduler

\begin{verbatim}
VAR: fired, fired_\gamma \in \{true, false\}
VAR: q_{\gamma_1}, q_{\gamma_2}, \ldots, q_{\gamma_n} \in Q \ Current cluster state for the n clusters
VAR: q_{a_1}, q_{a_2}, \ldots, q_{a_m} \in Q \ Current state for all dynamically scheduled actors
IN: The DFG g and its set of static actors \mathcal{A}_S \subseteq g.A
BEGIN
LET q_\gamma \leftarrow q_0 \ \forall g_\gamma \in g.G_\gamma
LET q_a \leftarrow q_0 \ \forall a \in g.A
DO LET fired \leftarrow false
FOREACH a \in g.A - \mathcal{A}_S DO
    WHILE \exists t \in a.T : t.q_{src} = q_a \land t.k DO
        CALL t.f\_action
        LET q_a \leftarrow t.q_{dst}
        LET fired \leftarrow true
ENDIF
ENDFOR
FOREACH g_\gamma \in g.G_\gamma DO
    DO LET fired_\gamma \leftarrow false
        FOREACH a \in g_\gamma.A DO
            WHILE \exists t \in a.T : t.q_{src} = q_a \land t.k DO
                CALL t.f\_action
                LET q_a \leftarrow t.q_{dst}
                LET fired_\gamma \leftarrow true
            ENDIF
        ENDFOR
    ENDIF
    LET fired \leftarrow fired_\gamma \lor fired
ENDWHILE fired
END
\end{verbatim}
This resulted in an overall execution time of about 2,230 ms for the same input stream using the dynamic scheduler, and 2,100 ms when using the QSSs. This corresponds to an improvement of approximately 6%. However, for data flow graphs containing static actors of fine granularity, the achievable reduction in overall overhead is expected to be higher.

![DFG of an MP3 decoder](image)

**Figure 4.58:** DFG of an MP3 decoder

The Motion-JPEG decoder (cf. Figure 4.59) has two static subclusters (the source actor being a trivial cluster containing only a single actor). As can be seen, the IDCT and the inverse ZigZag transformation correspond to the largest

**Algorithm 4** Rule subscheduler for clustered actors

```plaintext
... DO LET firedγ ← false
   FOREACH r ∈ R DO
      IF r.l ≤ qγ ≤ r.u ∧ r.kio THEN
         CALL r.f(...)
         LET qγ ← qγ + r.s
         LET firedγ ← true
      ENDIF
   ENDFOR
   IF qγ ≥ πI(qγ)(ηrepγ) THEN
      LET qγ ← qγ − πI(qγ)(ηrepγ)
   ENDIF
   LET fired ← firedγ ∨ fired
   WHILE firedγ
   ... 
```

189
4. Clustering

Each butterfly operation of the $8 \times 8$ inverse discrete cosine transform (IDCT) is modeled as an SDF actor (cf. Figure 4.10 on page 101). Such fine-grained graphs can result from the usage of tools like [VBCG04] which replace coarse-grained actors in a DFG by fine-grained subgraphs corresponding to DFGs derived from the functions of the actors. This expanded graph can then be clustered again to better utilize the resources of an MPSoC environment than would be feasible by only clustering actors atomically.

![DFG of a Motion-JPEG decoder](image)

**Figure 4.59:** DFG of a Motion-JPEG decoder

For test purposes, the standard Lena image was used and the decoding was repeated 25 times. For this test case, a reduction of 89% in the overall execution time of the decoder could be observed where the dynamic scheduler (M1) required 329 ms whereas after applying the presented rule-based QSS scheduling (M2) only 35 ms were needed. With the functionality restored, a reduction of 40% in the overall execution time of the decoder could still be observed where the dynamic scheduler (M1) had an overall execution time of 823 ms while only 492 ms were required by the rule-based QSS (M2).

In order to evaluate the presented clustering algorithm more thoroughly, it was applied to 48,000 randomly generated data flow graphs: The generated graphs have the same properties, except (1) the average input and output degree of the data flow graph vertices and (2) the number of actor firings necessary to reach the initial state of the SDF graph, i.e., the sum of the repetition vector scalars. The average input and output degree was varied from 2 to 7 in increments of 1 and the repetition vector sum from 100 to 1,000 in increments of 300. Using the SDF$^3$ tool [SGB06b], six different SDF graphs for each pair of average
input/output degree and repetition vector sum were generated, thus, creating 120 initial data flow graphs. The generated graphs consist of 60 actors and are cyclic, i.e., they contain strongly connected components, with random but consistent SDF rates and sufficient initial tokens to guarantee a deadlock-free self-scheduled execution. For each random graph $g$, various clusterings were generated. A clustering is a set of clusters $G_\gamma$ into which the static actors $A_S$ of the graph have been partitioned in such a way that each cluster conforms to the clustering condition given in Definition 4.4. To execute such clusterings, the dynamic scheduler from Algorithm 3 is simply equipped with the corresponding number of subschedulers (one for each cluster in the clustering). Furthermore, the number of dynamically scheduled entities $sched_{dyn}$ can be derived for each clustering as follows:

$$sched_{dyn} = |A - A_S| + |G_\gamma|$$

(4.66)

As one can see from the definition above, dynamically scheduled entities are dynamic actors and the composite actors (the $|G_\gamma|$ part) derived for all clusters. In fact, this exactly corresponds to the subschedulers in the dynamic scheduler and is also the number which is depicted on the x-axis of the speedup comparisons in Figure 4.61. For each of the six different initial SDF graphs corresponding to a pair of average input/output degree and repetition vector sum, a best clustering is selected for a given number of dynamically scheduled entities $sched_{dyn}$. The speedup value (on the y-axis) for a given average input/output degree $d$ (z-axis), number of dynamically scheduled entities $sched_{dyn}$ (x-axis) and repetition vector sum $r_{vs}$ (cf. Figures 4.61a to 4.61d) is the average of the six speedup values achieved by the best clusterings. The speedup is computed by dividing the throughput of the clustered DFGs by the throughput of the corresponding fully dynamic scheduled DFG. The throughput of a DFG is derived by dividing the number of iterations computed by the DFG by the overall execution time $t_{EXEC}$ required for the given number of iterations. In order to reduce the measuring inaccuracy of each measure, the number of iterations for a measure is adjusted in such a way that the overall execution time is between three and five seconds.

Frequently, the best clustering of a static region $A_S$ is not a monolithic composite actor $a_\gamma$, if such a cluster is feasible at all due to the clustering condition, but a partitioning of the static region into a set of smaller composite actors all having less complex input/output behavior. An example of such cases can be seen in Figure 4.60. Also note that the binary code size constraints of the automata-based approach (M3) forces the usage of more clusters in the clustering to satisfy the constraint of at most 133% increase in binary code size as compared to the fully dynamic scheduled system. This often yields a better speedup compared to a big composite actor.
To evaluate the improvement potential which can be exploited by the presented rule-based clustering approach, the overall overhead of the round-robin scheduler from Algorithm 3 was measured. Each initial data flow graph has per definition no dynamic actors. Hence, each initial data flow graph can be fully statically scheduled. The overhead $s_{\text{overhead}}$ is defined as the factor by which the fully statically scheduled initial data flow graph is faster than the same graph scheduled fully dynamically executing the same amount of useful work. Typically, thousands of iterations of the graph are used for these measurements. As clustering reduces this overall overhead, this represents an upper bound for the speedup that can be obtained by clustering.

However, if the clustering is between fully static and fully dynamic, the remaining overhead in the system can only be estimated. The estimation assumes that the overhead is uniformly spread over all actors:

$$s_{\text{estimation}} = \frac{s_{\text{overhead}} \cdot 59}{s_{\text{overhead}} \cdot (sched_{\text{dyn}} - 1) + 60 - sched_{\text{dyn}}}$$

Note that this is only a coarse approximation as the actors may have different repetition counts. As can be seen in Figure 4.61, for higher repetition vector sums, the approximation gets worse. This is expected as the speedup of the rule-based approach (M2) and the automata-based approach (M3) is the average of the best clustering for each random test graph. And the best clustering tends to select the actors for clustering that eliminate the most overall overhead from the system and not the average case assumed by the uniform spread of the overall overhead over all actors.

The available amount of memory is typically constrained in embedded systems. Hence, clusterings which, when synthesized by the rule-based approach (M2) or the automata-based approach (M3), exceed a binary code of of 133% of the code size of the dynamic case (M1) where excluded from consideration for

\[ \text{Figure 4.60: Example of the average clusters in the fastest clustering under binary code size constraints for random graphs with a repetition vector sum of 100} \]
Figure 4.61: Given above is the average speedup of the best clustering normalized to the throughput of the fully dynamically scheduled system for each of the six random test graphs with the given average input/output degree, repetition vector sum, and number of dynamically scheduled entities in the clustering. Furthermore, the binary code sizes produced by the synthesizing the clustering with the rule-based approach (M2) and the automata-based approach (M3) were also required to not exceed 133% of the code size of the dynamic case (M1).
4. Clustering

![Graphs showing normalized binary code size](image)

(a) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 100

(b) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 400

(c) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 700

(d) Average normalized binary code size of the best clustering for synthetic SDF graphs with a repetition vector sum of 1,000

**Figure 4.62:** Shown above is the average binary code size normalized to the code size of the fully dynamically scheduled system of the best clustering for each of the six random test graphs with the given average input/output degree, repetition vector sum, and number of dynamically scheduled entities in the clustering. No code size limitations where applied to find the clusterings with the best speedup.

and corresponds to 1. One can see a correlation between Figure 4.62 and Figure 4.61. In cases where the binary code size of the automata-based approach (M3) far exceeds the binary code size of the dynamically scheduled system (M1), the speedup of the automata-based approach (M3) under code size limitations also falls noticeably behind the speedup of the rule-based approach (M2). This blowup stems from the fact that the representation of QSSs via FSMs typically requires huge code sizes for complex schedules.
4.5.2 Design Space for Clustering

In the following, the potential of the presented clustering methodology and of monolithic clustering to reduce the number of dynamically scheduled entities is evaluated. While the best clustering in terms of speedup of a set of static actors may not be the clustering with the least number of dynamically scheduled entities, the design space for clustering supported by an approach comprises the span from the minimal number of dynamically scheduled entities given in Figure 4.63 to the fully dynamic case. Hence, the lower the obtainable number of dynamically scheduled entities for an approach is, the larger is the design space that is covered by this approach. A large design space is desirable to cover the conflicting requirements (cf. Section 4.1) of computational efficiency, e.g., from schedule overhead reduction, minimal latency, or overall throughput induced by the various architectural choices for embedded systems.

The test scenario consists of 240 SDF graphs that have been generated using the SDF\(^3\) tool [SGB06b] to obtain graphs varying in the average actor input/output degree from 2 to 7 in increments of 1 and in the repetition vector sum from 100 to 1,000 in increments of 300. All of these graphs have exactly 100 actors. For each pair of average input/output degree and repetition vector sum, 10 different SDF graphs have been generated. Of course, all of these SDF graphs can be fully statically scheduled. Hence, if none of the actors in these graphs are marked as dynamic, then both monolithic clustering as well as the automata-based approach can reduce the number of dynamically scheduled entities to one (cf. Figure 4.63a).

The compared approaches are monolithic clustering, which generates SDF composite actors and is labeled as SDF in Figure 4.63, and the presented clustering methodology in three different variants that are labeled as FSM-500, FSM-10,000, and heuristic. These three variants differ in the maximum permissible complexity of the QSSs realized by them: The variant labeled as heuristic corresponds to the number of dynamically scheduled entities obtained by the clusterings derived by the heuristic introduced in Section 4.4.1. Hence, the only constraint on clusters contained in these clusterings are a finite cluster state space. However, these clusterings might, in fact, not be implementable due to the huge binary code size requirements of complex QSSs. Thus, the variants labeled FSM-500 and FSM-10,000 are further constrained to only contain clusters with a cluster state space of cardinality less than 500 respectively 10,000 states.\(^{81}\) Hence, the minimal number of dynamically scheduled entities obtainable by the different approaches will always increase starting from the heuristic, to the FSM-10,000, FSM-500, and SDF approach.

\(^{81}\)Monolithic clustering can also be thought of as clustering with the constraint that each cluster has a cluster state space containing exactly one cluster state.
4. Clustering

Figure 4.63: Average number of dynamically scheduled entities obtainable by the four approaches for different DFGs

To evaluate the approaches in the presence of dynamic actors, a percentage of randomly selected actors have been marked as dynamic and, thus, unavailable for clustering. In detail, for each of the percentages 1%, 3%, 5%, 7%, and 9% (cf. Figures 4.63b to 4.63f), 10 variants have been derived from the 240 original SDF graphs by marking a random selection of the corresponding percentage of actors as dynamic. Hence, for each pair of average input/output
degree, repetition vector sum, and percentage of dynamic actors, 100 graphs are used to average the number of dynamically scheduled entities obtainable by the four approaches. The number of dynamically scheduled entities for the heuristic approach are derived by applying the heuristic from Section 4.4.1 to each of the 100 graphs, computing the number of dynamically scheduled entities (cf. Equation (4.66)) for the resulting clusterings, and averaging the 100 results. For the approaches $SDF$, $FSM-500$, and $FSM-10,000$, the clusters in the clustering derived from the heuristic are further divided until the corresponding constraint on the maximal cardinality of the cluster state space is satisfied.

When considering Figure 4.63 as a whole, one can observe that even the presence of one dynamic actor (cf. Figure 4.63b) in the DFG drastically limits the potential of monolithic clustering. For graphs with an average actor input and output degree of two, already around fifty entities must be dynamically scheduled, whereas the proposed clustering methodology only requires around two dynamically scheduled entities. Furthermore, with increasing average input and output degree, the complexity of the required QSSs for the clusters increases. Thus, smaller clusters must be used to conform to the cluster state space cardinality constraint and, hence, the number of required dynamically scheduled entities increases. A further contributing factor to complexity is the repetition vector sum of the graph. An increased repetition vector sum will increase the set of input/output dependency function values (cf. Definition 4.8 on page 139) from which the cluster FSM is derived. This complexity factor is dominant for large clusters with few FIFO channels connecting them to their cluster environment. These types of clusters are present if few dynamic actors are contained in a DFG (cf. Figures 4.63a to 4.63c). In contrast to this, if the number of dynamic actors and the average actor input and output degree increases (cf. Figures 4.63d to 4.63f), then the number of FIFO channels connecting clusters to their cluster environment will also increase. Then the dominant complexity factor is the required interleavings of output actor firings (cf. Definition 4.10 on page 141). As can be seen in Figures 4.63d to 4.63f, clusters containing marked graphs\(^{82}\) are especially susceptible to this phenomenon.

To exemplify, the automata- and rule-based approaches have been applied to the 100 actor test graphs used in this section. The considered clusterings have been constrained by the requirement that their implementations have a binary code size of at most 133\% of the fully dynamically scheduled implementation (M1) of the corresponding graph. As can be seen in Figure 4.64, the rule-based approach (M2) has a higher potential to reduce the number of dynamically scheduled entities in comparison to the automata-based approach (M3). This

\(^{82}\)The SDF graphs used in this section have exactly 100 actors. Hence, if the repetition vector sum of a graph is also 100, then each actor must fire exactly once in an iteration of the graph. Thus, the graph is a marked graph or a graph where the consumption and production rates for each edge of the graph are identical.
is especially evident in the situations (cf. Figures 4.64d to 4.64f) where the complexity of the QSSs stems from the required interleavings of output actor firings.

**Figure 4.64:** Average number of dynamically scheduled entities obtainable under 133\% code size limitations by the automata- and rule-based approaches
4.6 Related Work

The considered refinement algorithms can be distinguished into (1) algorithms for clustering SDF graphs into monolithic clusters, that is each monolithic cluster can again be represented by an SDF composite actor, and (2) algorithms for clustering an SDF or CSDF graph into composite actors each of which can be represented not by an SDF or CSDF actor, but by a composite actor implementing a QSS. The algorithms in (1) are applicable to closed systems, that is working on whole SDF graphs. The algorithms in (2) are developed for open systems, that is working on clusters potentially having inputs and outputs to an unknown cluster environment.

The first set of clustering algorithms for generating monolithic clusters has mainly been researched by Bhattacharyya et. al. [BL93, PBL95]. However, these algorithms severely constrain the possibilities of clustering an SDF graph. Furthermore, the Pairwise Grouping of Adjacent Nodes (PGAN) algorithm presented in [BL93] does only work for SDF graphs and has at its main focus the reduction of code and data memory for uniprocessor systems. The heuristic presented in [PBL95] also works for SDF subgraphs, that is static clusters, embedded in dynamic DFGs when both constant consumption and production rates are known for each edge connecting the dynamic DFG to the SDF subgraph. However, the heuristic is still limited to generate only monolithic clusters, hence, it is severely constrained in the clustering possibilities.

The second set of refinement algorithms for generating composite actors implementing QSSs has mainly been researched in this thesis [FKH*08*, FZK*11*, FZHT11*, FZHT13*] and by Tripakis et al. [TBG*13]. The algorithms presented in [FZHT11*, FZHT13*, FZK*11*] generate basically the same QSS for a static cluster. The only difference between [FZK*11*] (cf. Section 4.4.2) and [FZHT11*, FZHT13*] (cf. Section 4.4.3) is the representation of the QSS.

In contrast to this, Tripakis et al. [TBG*13] considers cluster refinement under the aspect of modular code generation for SDF graphs. Both approaches lack information about the cluster environment of a cluster. In the case of the cluster environment being a dynamic DFG, this lack of information is due to the lack of analyzability of dynamic DFGs in the general case. In case of modular code generation, the cluster environment is not known at the time of cluster refinement. Note, however, that modular code generation only uses QSSs to preserve scheduling flexibility for the intermediate representations used by successive compilation steps that successively assemble actors and clusters into a whole SDF graph. The final whole SDF graph will be scheduled by a traditional PSOS that no longer contains any actors implementing a QSS.

More precisely, the approach of Tripakis et al. uses so-called Deterministic SDF with shared FIFOs (DSSF) graphs as input and output for the presented modular code generation algorithm. The DSSF model extends SDF in such
a way that even when shared FIFOs are used in DSSF, determinism can be ensured. Since DSSF is only an extension of SDF, it enables reusage of many techniques developed for standard SDF. More importantly, the QSS representation chosen in [TBG+13] is a DSSF subgraph that is called an SDF profile. Modular code generation is concerned with the generation of an SDF profile from an (original) DSSF subgraph. This process can reduce the number of actors in the DSSF subgraph representing the SDF profile compared to the original DSSF subgraph. The reduction is facilitated by combining the firings of multiple actors of the original DSSF subgraph into one or a lesser number of actors of the SDF profile. The ability of an SDF profile to partition the firings of one actor of the original DSSF subgraph required for one iteration of the original DSSF subgraph amongst multiple actors of the SDF profile enables a larger design space for cluster refinements as compared to monolithic clusterings representable by an SDF subgraph. However, the data flow characteristics of the DSSF model constrain the QSSs representable by SDF profiles to always be cyclic. While the DSSF model can represent QSSs with multiple cycles, QSSs containing initial retiming sequences cannot be represented by SDF profiles. Moreover, Tripakis et al. does not need to tackle the FIFO size adaption (cf. Section 4.4.6) problem since only SDF cluster environments are considered by the approach. Hence, determining the channel capacities in order to prevent deadlocks or satisfy various other criteria is a decidable problem [BMMKU10] that can be performed at compile time while the final whole SDF graph is assembled.

In contrast to [Buc93, SLWSV99], which represent a monolithic approach to schedule dynamic DFGs, the presented clustering methodology is inherently working on static islands which represent data flow subgraphs. Hence, the algorithms from [Buc93, SLWSV99] have an upper problem size to be computationally feasible, while clustering can handle data flow graphs of arbitrary size by only considering static subgraphs of sizes that are computationally feasible for the presented clustering methodology. Of course, optimization potential is lost if individual parts of the static island in the DFG are scheduled in isolation by the clustering methodology.

Summary

In this chapter, the second key contribution of this work, a clustering methodology [FKH+08*, FZK+11*, FHZT13*] that exploits islands of static actors in more general DFGs, has been presented. Clustering replaces islands of static actors in the data flow graph by composite actors. Two variants of clustering, an automata-based and a rule-base approach, are contributed. The rule-based representation trades latency and throughput in order to achieve a more compact code size of a QSS as compared to the automata-based representation. However,
under code size constraints, the rule-base approach might offer a better latency and throughput due to its ability to represent clusters containing a larger number of static actors. Later, in the implementation generated by the synthesis back-end, the remaining actors which are not replaced by a composite actor as well as all composite actors are scheduled by a dynamic scheduler at run time. Thus, by combining multiple static actors into one composite actor, the clustering methodology reduces the number of actors which have to be scheduled by the dynamic scheduler. This reduction is of benefit as it has been shown that compile-time scheduling of static actors produces more efficient implementations in terms of latency and throughput than run-time scheduling of the same actors by a dynamic scheduler.

Furthermore, clustering enables the generation of QSSs that employ sequences of statically scheduled actor firings. Hence, clustering not only reduces the scheduling overhead imposed by the run-time system to find a schedule of all actors of the DFG at run time, but clustering also enables efficient code generation for sequences of statically scheduled actor firings as compared to code generation optimizations possible if all actors of a sequence are considered in isolation by a compiler.

In contrast to known clustering methods from literature [KB06, CH97, BML97, PBL95], the presented methodology is not limited to clustering only SDF graphs, and neither limited to produce only SDF actors from static islands. Hence, compared to methods from literature [KB06, CH97, BML97, PBL95], the presented approach drastically enlarges the possible applicability to heterogeneous DFGs with static and dynamic actors. Thus, the ability to generate QSSs instead of only SDF or CSDF actors enables the generation of optimized schedules for a much larger variety of static islands.
In this chapter, the third key contribution of this work, a synthesis back-end for the SystemC Models of Computation (SysteMoC) language, will be presented. The synthesis back-end contributes to the realization of the promised productivity gain [FZH+10*] of Electronic System Level (ESL) modeling by providing an automatic way [ZFS+14*] to generate virtual prototypes from SysteMoC applications. Here, the synthesis back-end supports multiple different targets like pure single threaded C++ software implementations, multithreaded C++ software implementations, software for the Central Processing Units (CPUs) employed in a Multi-Processor System-on-Chip (MPSoC) system, as well as SystemC code suitable as input for commercial behavioral synthesis tools for the generation of hardware accelerators. The synthesis back-end employs target dependent source-to-source transformations that can transform a SysteMoC actor into an appropriate source format for these different targets.

In general, the synthesis back-end [ZHFT12*, ZFHT12*, ZFS+14*] generates an implementation by transforming a SysteMoC application according to a set of design decisions taken by the SYSTEMCoDESIGNER [HFK+07*, KSS+09*] Design Space Exploration (DSE) methodology. Examples for design decisions are allocation and binding decisions, as well as the Quasi-Static Schedules (QSSs) generated by the clustering methodology. The allocation defines the set of hardware resources that are used in an implementation, and the binding provides the information which SysteMoC actor is implemented on which resource. Note that the selection of a concrete realization for each communication channel is also performed during DSE. The QSSs generated by the clustering methodology are used by the synthesis back-end to reduce the scheduling overhead of the generated software implementations running on the CPU resources of the overall MPSoC system.

First, the synthesis back-end is used to derive the characteristics, e.g., execution delay, binary code size, etc., of a single actor when implemented on the various possible targets. These characteristics are required by the SYSTEMCoDESIGNER DSE methodology to evaluate the candidate implementations at ESL. Moreover, the synthesis back-end is used to realize synthesis-in-the-loop for the SYSTEMCoDESIGNER DSE methodology: Synthesis-in-the-loop enables
the DSE to evaluate its design decisions based on real implementations, in con-
trast to the approximately-timed performance simulation carried out via the 
Virtual Processing Components (VPC) [SFH+06*] approach at ESL. As a com-
promise between the evaluation accuracy obtainable by the synthesis-in-the-loop 
approach and the evaluation speed obtained via the VPC approach, synthesis-
in-the-loop could also be used to consolidate the characteristics for possible im-
plementations of whole groups of actors on a resource in contrast to the modular 
analysis from the individual characteristics of the constituting actors performed 
at ESL.

The rest of this chapter is structured as follows: First, the presented approach 
is motivated in Section 5.1. Then, the integration of the synthesis back-end into 
the SYSTEMCoDESIGNER design flow is shown in Section 5.2. In Section 5.3, the 
Motion-JPEG decoder introduced in Chapter 4 is briefly recapitulated. In order 
to derive an implementation at a lower level of abstraction from a SysteMoC 
application at the ESL, source code transformations have to be performed. Ex-
amples for these transformations are demonstrated in Section 5.4. In Section 5.5, 
results from applying the presented synthesis back-end to the Motion-JPEG de-
coder example are given. Finally, the related work is reviewed in Section 5.6. 
Note that parts of this chapter are derived from [HFK+07*, KSS+09*].

5.1 Motivation

Several commercial C/C++/SystemC-based behavioral synthesis (also known as 
high-level synthesis) tools exist, e.g., *Forte Cynthesizer* [For13], *Catapult C* from 
Calypto Design Systems [Cal13], or Vivado from Xilinx [XIL14]. These tools 
allow the automatic translation of behavioral SystemC models to the Register 
Transfer Level (RTL). However, synthesis of an entire system using behav-
ioral synthesis tools is either not possible at all or prohibitive due to the sheer 
complexity of these systems.

To overcome these limitations, the proposed synthesis back-end integrates behav-
ioral synthesis into the SYSTEMCoDESIGNER DSE methodology. That way, 
it becomes possible to automatically (1) optimize MPSoC implementations and 
(2) generate the resulting MPSoC implementation. The synthesis back-end sup-
ports the generation of pure software, as shown in Section 4.5 for the results 
of the single processor evaluation of the clustering methodology that have been 
generated by measuring the resulting throughput of the implementations gen-
erated by the presented synthesis back-end. Moreover, the generation of virtual 
prototypes for hardware/software implementations [ZFS+14*] is also possible. 
However, the focus of this chapter is on the generation of pure hardware imple-
mentations, where an automatic behavioral synthesis step is used to generate 
RTL code for single actors (SystemC modules).
In particular, the integration of Forte Synthesizer into the SystemCoDesigner DSE environment will be highlighted. The target architecture is a Xilinx Virtex-II Field Programmable Gate Array (FPGA). Here, Synthesizer is used to automatically generate synthesizable RTL code from SystemC modules. The resulting actor implementations are characterized regarding their size (number of Look-Up Tables (LUTs), number of Block Random Access Memory (BRAMs), etc.) and their performance (delay). Note that it is possible to generate different implementations for each actor with respect to size and performance. The attributes of the actor implementations are used during DSE to select an implementation for each actor, resulting in a set of non-dominated solutions. From this set, the designer can select an implementation of the entire system and synthesize it using Xilinx Embedded Development Kit (EDK) [XIL05]. For this synthesis, a library providing implementations for the communication primitives is supported. In order to facilitate this integration, the synthesis back-end is used to translate each SysteMoC actor into a SystemC module.

5.2 Synthesis in the SystemCoDesigner Design Flow

The overall design flow of the SystemCoDesigner DSE methodology is based on (1) actor-oriented modeling in SysteMoC, as presented in Chapter 3, (2) hardware generation for each actor using translation of SysteMoC actors to SystemC modules and subsequent behavioral synthesis, (3) DSE to find the best candidate implementations, and (4) automatic generation of the MPSoC by the presented synthesis back-end. With the help of Figure 5.1, the required steps to integrate the synthesis back-end into the SystemCoDesigner design flow are briefly summarized.

Note that the parts of the design flow contained in the dashed box have already been presented in Chapter 3. As introduced, the design flow starts by modeling an application as an executable specification using the SysteMoC library. Note that the SysteMoC application can be functionally simulated for testing purposes.

Next, the SysteMoC infrastructure is used to extract the network graph of the application. An exact definition of a network graph is given in Definition 3.3 on page 49. Note that for historical reasons, the name problem graph [BTT98] is commonly used if this graph is considered from the DSE perspective. Then, the synthesis back-end is used to transform each SysteMoC actor into a SystemC module. A brief outline of this transformation is given in in Section 5.4. The resulting SystemC module is used as input for the behavioral synthesis tool.

From a broader DSE perspective, a problem graph may not necessarily exhibit data flow semantics and might only encode some kind of data communication. In contrast to this, the name network graph is used if data flow aspects are in focus.
Moreover, multiple possible RTL implementation can be generated from the same SystemC module by varying the area and latency constraints given to the behavioral synthesis tool. Each of these variants is added to the component library and their characteristics, e.g., execution delay and area consumption, is derived from the synthesis reports of the behavioral synthesis tool.

In the next step, an architecture graph [BTT98] is formed by using the modules of the component library as resources. Moreover, other modules such as CPUs, Intellectual Property (IP) cores, hand-optimized modules, and communication channels and buses, etc., can be used to derive the architecture graph. More formally, an architecture graph \( g_a = (R, L) \) is composed of resources (CPUs, buses, etc.) modeled by vertices \( R \) and communication links modeled as edges \( L \) between these resources. Furthermore, mapping edges \( M \subseteq A \cup C \times R \) will be added for each actor of the network graph to their possible hardware implementations derived via behavioral synthesis. If a CPU is present in the architecture
5.3 Case Study

For automatic DSE and automatic behavioral synthesis, only a well defined subset of SystemC is suitable. Here, SysteMoC [FHT06\*] is used to cover both the needs of DSE and automatic behavioral synthesis. Communication is restricted to dedicated channels. Actors contain ports, to which the channels are connected. Here, port-to-port communication media with FIFO semantics that connect exactly one output port with exactly one input port are assumed.
5. Synthesis

Figure 5.2: Shown above is a DFG of a Motion-JPEG decoder, transforming a Motion-JPEG data stream into uncompressed images. The DFG consists of 9 actors connected amongst each other via 12 FIFO channels.

See Figure 5.2 for the simplified SysteMoC model of the Motion-JPEG decoder, which was introduced previously in Chapter 4, that is also used as running example throughout this chapter. The Motion-JPEG decoder case study consists of 5,650 lines of code, supporting interleaved and non-interleaved baseline profile without sub sampling. It transforms a baseline Motion-JPEG stream from the JPEG Source actor into a series of Portable Pixmap (PPM) images that are written by the ImageSink actor. The parser extracts and propagates the control information like the image size and quantization steps. After entropy decoding (Huffman Decoder) and block reconstruction (Inverse ZRL to Inverse ZigZag), an Inverse Discrete Cosine Transform (IDCT) is performed. The Frame Shuffler reorders the pixels of the resulting 8×8 blocks into a raster scan order which are sent to the ImageSink actor.

Each of these actors is modeled in SysteMoC and is divided into three parts: (1) the communication ports, (2) the actor functionality, and (3) the actor communication behavior. Actors may posses some internal state (internal variables), e.g., as demonstrated by the ImageSink SysteMoC actor shown in Figure 5.3.

Figure 5.3: Visualized above is the ImageSink actor from Figure 5.2. The actor has two input ports \( i_1 \) and \( i_2 \) as well as an actor FSM that determines its communication behavior. The FSM consists of two states and three transitions. Each transition is of the form \( \text{guard} / \text{action} \).
5.4 Synthesis of Actor-Oriented Designs

Actor functionality is a collection of functions that can access data on channels via ports (e.g., processPixel() in Figure 5.3). These functions are only executed during transitions of the actor FSM which implements the communication behavior of the actor. Transitions in this FSM include a guard and an action function, which is executed if the transition is taken. The guard determines under which conditions a transition may be taken, e.g., depending on a certain minimal amount of input tokens or values of tokens on input ports. A transition can only be taken, if the guard is satisfied and therewith activate the transition. For example, see the transition from state $q_{\text{write}}$ to state $q_{\text{start}}$ in Figure 5.3.

Separated by a logical conjunction, there are two parts in this guard. The first part, $\#i_2 \geq 1$, demands at least one available token at input port $i_2$ which is consumed (i.e., removed from the channel) after execution of the action; the second part, $m_p = 1$, asserts that some internal variable $m_p$ equals '1'. If both parts are satisfied, the transition is activated. Taking the transition leads to the execution of the action function $f_{\text{processPixel}}(i_2[0])$ (which possibly changes internal variables), consuming and processing the first token from the channel connected to port $i_2$, and a state transition from state $q_{\text{write}}$ to $q_{\text{start}}$. A precise definition of this notions is given in Chapter 3.

5.4 Synthesis of Actor-Oriented Designs

In this section, the automatic transformation of SysteMoC actors to pure SystemC modules for behavioral synthesis, as well as the automatic system generation and the used communication primitives are described. The automatic transformation is performed by a source-to-source compiler contained in the synthesis back-end. This source-to-source compiler is based on the C Language Family Frontend of the LLVM Compiler Infrastructure (clang) [Lat11]. The transformed SystemC description is synthesizable to Verilog Hardware Description Language (Verilog) using Forte Cynthesizer, a state-of-the-art behavioral synthesis tool. For each synthesized Verilog RTL module, the makespan for the execution of each action are extracted by means of analyzing the synthesis reports provided by Cynthesizer. These timing values are used for optimizing the latency during DSE with SystemCoDesigner. In the automatic system generation phase, the synthesized Verilog RTL modules are being connected and synthesized according to the DSE results to obtain an FPGA implementation.

5.4.1 Actor Transformation

SysteMoC actors contain abstract communication ports, a dedicated actor FSM, and action and guard functions. Whereas SystemC modules are C++ classes containing ports for hardware signals and parallel processes. To transform the
abstract SysteMoC communication ports to ports for hardware signals, the accesses to these ports are replaced and an FSM process controlling the guard and action invocations is created. The transformation steps [KSS+09*] are depicted in Figure 5.4.

For each SysteMoC communication port, hardware signal ports are created to be able to connect the hardware implementation of SysteMoC FIFOs as described in Section 5.4.3. There are processes for each SysteMoC write port that handle the hardware protocol, and, thus, allow concurrent writes to different hardware ports. Concurrent reads from different input ports are realized by splitting the read access into two functions: The first function starts the read access for a port and does not wait for a clock edge, while the second function waits for the result from the input port. Hence, reads from different input ports can be started concurrently by calling the first function for each input port.

To transform the action and guard functions, all port accesses are replaced by function calls that access the hardware ports as described above. The transformed functions are not implemented as parallel processes, but they are called from the controlling FSM process. To generate hardware alternatives which may be different in speed and size, the designer can place directives, e.g., constraints for the latency of a loop body, for the behavioral synthesis tool within the transformed SystemC code. These alternatives can be used during DSE to generate different individuals including trade-offs, e.g., area vs. throughput.
In SysteMoC, there is an abstract syntax to describe the actor FSM of each actor, which cannot be used in pure SystemC modules. To create a SystemC module, a controlling process containing the transformed actor FSM is automatically created. This process is the only process schedulable for the behavioral synthesis tool if concurrent guard evaluation is not enabled. Otherwise, in addition to the controlling process derived from the actor FSM, a guard process will be generated for each guard. Thus, the controlling process will not evaluate the guard functions, but only use the results computed by the concurrently executing guard processes.

The transformed actor FSM code uses a switch statement for the state variable. Within each state, all possible transitions are checked sequentially by checking the token counts in the input and output FIFOs and calling the guard functions. If a transition is taken, the corresponding action function is called, tokens are consumed from the input FIFOs and tokens are produced on the output FIFOs. Without optimizations, the possible transitions within each state are checked sequentially, because guards access input ports and, thus, cannot be parallelized without additional hardware resources. However, if additional hardware registers are used to cache tokens from input ports, then parallel guard evaluation [ZHFT12*] is possible.

5.4.2 Generating the Architecture

As depicted in Figure 5.1, the result of the automatic DSE is a set of Pareto-optimal individuals. From these individuals, the designer can select one as implementation according to strategic considerations. This process is known as decision making in multi-objective optimization.

The designer selects individuals resulting from the automatic DSE for implementation. These individuals are used as input to the synthesis back-end in order to generate the FPGA-based designs. In the following case study, the hardware-only system generation flow for Xilinx FPGA targets is described.

The architecture generation is twofold: In the first step, the synthesis back-end automatically inserts the allocated IP cores from the component library. In the second step, it automatically inserts the communication resources from the communication library. The result of this architecture generation is a hardware description file (.mhs-file in case of the EDK tool chain). In the following, the relevant details of the architecture generation process are discussed.

Beside the information stored in the architecture graph, information from the SysteMoC model must be considered during the architecture generation as well. A vertex representing a FIFO in the problem graph contains information about the depth and the data type of the communication channel used in the SysteMoC model. A vertex representing an Actor contains the names of its
5. Synthesis

<table>
<thead>
<tr>
<th>Operation</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd_tokens()</td>
<td>Returns how many tokens can be read from the SysteMoC FIFO. (available tokens)</td>
</tr>
<tr>
<td>wr_tokens()</td>
<td>Returns how many tokens can be written into the SysteMoC FIFO. (free places)</td>
</tr>
<tr>
<td>read(offset)</td>
<td>Reads a token at a given offset relative to the first available token. The read token is not removed from the SysteMoC FIFO.</td>
</tr>
<tr>
<td>write(offset, value)</td>
<td>Writes a token at a given offset relative to the first free place. The written token is not made available.</td>
</tr>
<tr>
<td>rd_commit(count)</td>
<td>Removes count tokens from the SysteMoC FIFO.</td>
</tr>
<tr>
<td>wr_commit(count)</td>
<td>Makes count tokens available for reading.</td>
</tr>
</tbody>
</table>

Table 5.1: [HFK+07*] SysteMoC FIFO interface

SysteMoC ports; these names are used for mapping SysteMoC communication ports to hardware ports.

After instantiating the IP cores, the final step is to insert the communication resources. These communication resources are taken from the platform-specific communication library. In the case of hardware-only systems, the used communication resource are always SysteMoC FIFOs as described in the next section.

After generating the architecture, several Xilinx implementation tools like map, par, bitgen, data2mem, etc., are used to produce the platform-specific bit file. Finally, the bit file can be loaded on the FPGA platform and the system can be run.

5.4.3 Communication Resources

The SysteMoC FIFO communication resources serve three main purposes: They store data, transport data, and synchronize the actors via availability of tokens respectively buffer space. For these tasks, the interface shown in Table 5.1 is used. In the following, each communication resource that implements the interface from Table 5.1 is called a SysteMoC FIFO.

The implementation of SysteMoC FIFOs is not limited to be a single hardware FIFO module. It may, e.g., consist of two hardware modules that communicate over a bus. In this case, one of the modules would implement the read interface, the other one the write interface.

To be able to store data in the SysteMoC FIFO, it has to contain a buffer. Depending on the implementation, this buffer may also be distributed over
different modules. The designer specifies the buffer sizes in the SysteMoC model. Of course, it would be possible to optimize the buffer sizes for a given system. However, this is future work in SystemCoDesigner.

As can be seen from Table 5.1, a SysteMoC FIFO is more complex than a common hardware FIFO. This is due to the fact that common hardware FIFOs do not support non-consuming read operations for guard functions and that SysteMoC FIFOs must be able to commit more than one read or written token.

The hardware implementation of SysteMoC FIFOs consist of a single module described in Verilog. This is a generic module that is synthesizable to efficient hardware implementations on different FPGA platforms. For example, the buffer containing the FIFO data can either be composed of BRAM cells or provided by distributed RAM on Xilinx FPGAs. As shown in [HFK+07*], the overhead compared to native Xilinx COREGEN FIFOs is rather small, e.g., for a SysteMoC FIFO providing storage in 8 BRAMs for 4,096 tokens each of which has a data width of 32 bit, the overhead is just 12 FFs and 33 4-input LUTs.

5.5 Results

The presented design flow is used to create a hardware implementation of a Motion-JPEG decoder as depicted in Figure 5.2. The target platform is a Xilinx Virtex-II FPGA (XC2V6000) where the designer has chosen a target clock frequency of 50 MHz.

First, an executable specification of the Motion-JPEG decoder using SysteMoC. Then, the actor specifications are automatically transformed to synthesizable SystemC modules. These SystemC modules were synthesized to a gate-level netlist using Forte Design Systems Cynthesizer [For13] and Synplify Pro from Synopsys [Syn13].

The specification graph provided at least one hardware implementation alternative for each actor. For three actors, two hardware implementation alternatives were provided for each of them. Two alternatives for SysteMoC hardware FIFOs implementations were available. One alternative uses BRAM, the other uses distributed RAM to provide the storage for the tokens. The used LUTs, FFs, BRAMs, and Hardware Multipliers (HMULs) for each actor were automatically extracted from the synthesis reports. Also the latency information for each action of each actor was automatically extracted from the high-level synthesis report. All this information was used during DSE. The DSE uses VPC with action accurate simulation to get latency and throughput data [KSS+09*].

Table 5.2 shows selected properties of individuals found by DSE. The performance values latency and throughput are VPC simulation results for QCIF (176x144 pixel) pictures. The first individual was also synthesized for the given
5. Synthesis

<table>
<thead>
<tr>
<th>Latency</th>
<th>Throughput</th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAMs/HMULs</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.35 ms</td>
<td>98.54 fps</td>
<td>36 059</td>
<td>15 885</td>
<td>79</td>
</tr>
<tr>
<td>10.32 ms</td>
<td>98.60 fps</td>
<td>45 250</td>
<td>16 601</td>
<td>53</td>
</tr>
<tr>
<td>10.32 ms</td>
<td>98.61 fps</td>
<td>41 963</td>
<td>16 401</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 5.2: VPC simulation results

<table>
<thead>
<tr>
<th>Latency</th>
<th>Throughput</th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAMs/HMULs</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.45 ms</td>
<td>43.5 fps</td>
<td>36 000</td>
<td>15 800</td>
<td>79</td>
</tr>
</tbody>
</table>

Table 5.3: Hardware synthesis results of the first design point from Table 5.2

The used hardware resources are very close to the estimated values, because the estimation already uses data from the gate-level synthesis. During VPC simulation, time passes only while executing actors, not in the controlling actor FSM or guard functions. This is the reason for the discrepancy between the VPC and RTL simulation performance results. To minimize this discrepancy, two things can be done: (1) Parallel guard evaluation [ZHFT12*] can be performed in hardware. This will minimize the hardware overhead in the controlling FSM process. (2) The scheduling overhead spent by the controlling FSM process can be distributed to the action execution times by means of calibration [KSH+11, KSH+12].

5.6 Related Work

In this section, related work in the area of DSE at the ESL as well as automatic synthesis of systems is discussed. Tools providing DSE at the ESL are for instance Sesame [PEP06], MILAN [MPND02], and CHARMED [KB04]. However, all these tools do not provide a synthesis path to generate an implementation. On the other hand, some tools exist providing synthesis to hardware/software implementations but do not perform an automatic DSE. For instance, ESPAM/Compaan/Laura [DTZ+08, NSD08] automatically converts a Matlab loop program into a Kahn Process Network (KPN). This process network can be transformed into a hardware/software system by instantiating processors and IP cores and connecting them with FIFOs. Special software routines take care of the hardware/software communication. The EDK tool chain is used for final bit stream generation.

The main difference between ESPAM/Compaan/Laura and SystemCoDESIGNER lies in the supported Models of Computation (MoCs). The Com-
paan/Laura approach [SZT+04] restricts the MoC to KPNs, while SYSTEM-CODESIGNER supports Non-Determinate Data Flow (NDF). Moreover, ES-PAM does not provide a way to use behavioral synthesis.

The Center for Embedded Computer Systems [Cen] has introduced its Embedded Systems Environment (ESE). ESE starts with a SystemC description and synthesizes an SoC implementation. In a first step, the mapping of operations to resources is done manually. In a second step, a SystemC transaction level model is generated which is used for synthesis. Here, ESE provides a behavioral synthesis approach to generate hardware implementations from SystemC modules. However, in contrast to SYSTEM-CODESIGNER, the automatic DSE is excluded, i.e., the mapping has to be done manually.

An approach supporting automatic DSE as well as synthesis is presented in [KKO+06]. It is called Koski and, as SYSTEM-CODESIGNER, it is dedicated to the automatic SoC design. The input specification is given as a KPN modeled in UML. The Kahn processes are refined using Statecharts. The target architecture consists of the application software, the platform-dependent and platform-independent software, and synthesizable communication and processing resources. Moreover, special functions for application distribution are included, i.e., inter-process communication for multi-processor systems. During DSE, Koski uses simulation for performance evaluation. Although, many similarities can be identified between Koski and SYSTEM-CODESIGNER, there are fundamental differences: Koski in its current version provides a sophisticated software generation targeting real-time operating systems allowing a great flexibility. On the other hand, SYSTEM-CODESIGNER integrates behavioral synthesis tools into the DSE removing any manual step from the design flow.

Summary

In this chapter, the third third key contribution of this work, a synthesis back-end for the SysteMoC language, is presented. The integration of this synthesis back-end with the SYSTEM-CODESIGNER DSE methodology allows the automatic optimization and generation of MPSoC systems—starting with an abstract actor-oriented model written in SysteMoC. The presented design flow allows the generation of MPSoC implementations, determined by means of DSE, in very short time. Furthermore, the presented synthesis back-end can be used to realize synthesis-in-the-loop for the SYSTEM-CODESIGNER DSE methodology. This enables SYSTEM-CODESIGNER to evaluate its design decisions based on real implementations, in contrast to the approximately-timed performance modeling performed via the VPC [SFH+06*] approach. Additionally, the implementations generated via the synthesis back-end can be used to validate the annotated timing values [SGHT10] (cf. the discrepancies between the estimated
and the measured latency in Tables 5.2 and 5.3) used for the VPC-based back annotation of design decisions into the SysteMoC application for the purpose of simulative evaluation at the ESL.

Moreover, by automatic generation, the synthesized system avoids errors that might be introduced by manual refinement steps. Furthermore, no manual optimizations are made. Instead, clustering is used to optimize the schedules for actors bound onto the same resource. Finally, the efficiency of the presented design flow was demonstrated using a Motion-JPEG decoder as an industrial grade case study.
Outlook and Conclusions

The main contributing factor for the complexity explosion of modern embedded Systems is the heterogeneous architecture of Multi-Processor System-on-Chip (MPSoC) systems used to realize these embedded systems. This rising complexity is increasingly threatening the feasibility of traditional embedded system design flows. Hence, the adoption of design methodologies at the Electronic System Level (ESL) becomes imperative. To cope with the concurrency implied by these architectures, data flow modeling is a widely accepted paradigm [LNW03, LN04] at the ESL. This work proposes a clustering-based MPSoC design flow for data flow-oriented applications for MPSoCs. The introduced approach aims at a seamless design flow from an application modeled at the ESL down to a virtual prototype or a Field Programmable Gate Array (FPGA) based implementation. It puts special emphasis on efficient code generation for data flow graphs of very fine granularity. In the following, the key contributions of this thesis are summarized. Then, an outlook on future research perspectives is given.

6.1 Summary

This thesis contributes improvements for design flows targeting MPSoC systems by: (1) providing the SystemC Models of Computation (SysteMoC) modeling language with formal underpinnings in data flow modeling to enable a trade-off between analyzability and expressiveness of the described application; (2) contributing a methodology called clustering that exploits these trade-offs to generate efficient schedules for applications containing actors of high analyzability and low expressiveness; and (3) enabling the synthesis of the modeled applications via a synthesis back-end that supports multiple targets such as single core software implementations, virtual prototypes of distributed MPSoC systems, and dedicated hardware implementations.

Every contribution of this thesis is essential for the proposed design flow. In the following, these steps and their importance will be reiterated. The first key contribution of this work—the SysteMoC modeling language introduced in
Chapter 3—is required to serve as design entry for the presented design flow. The SysteMoC language is used for modeling of executable specifications at the ESL. Other languages like Cal Actor Language (CAL) [EJ03] or pure SystemC [GLMS02] can also be used to model executable specifications, but lack the integration into a Design Space Exploration (DSE) methodology. To facilitate this integration, this thesis contributes both a way to extract a data flow graph from a SysteMoC application as well as a way to enable the DSE to back annotate design decisions into the SysteMoC application for the purpose of simulative performance evaluation.

If an executable specification is modeled as a coarse-grained data flow graph, a naïve translation of one actor to one process, thread, or isolated unit of compilation is feasible for translation of this executable specification into an implementation. However, if the data flow graph is very fine grained, the execution time required for synchronization and handling of data dependencies between these actors at run time will dominate the amount of useful computational work performed by the actors. That is to say, for very fine-grained data flow graphs the scheduling overhead will dominate the amount of useful computational work.

This problem can be mended by coding the actors of the application at an appropriate level of granularity, i.e., combining as much functionality into one actor such that the computation performed by the actor dominates the execution time required for scheduling. However, combining functionality of multiple actors into one actor can also mean that more data has to be produced and consumed atomically which requires larger channel capacities and may delay computation unnecessarily. This degrades the latency if multiple computation resources are available for execution of the application. Hence, selecting a level of granularity for a functionality by the designer is a manual trade-off between schedule efficiency to improve throughput and latency on a single core software implementation against scheduling flexibility required to improve throughput and latency on an MPSoC implementation. Thus, if the mapping of actors to resources itself is part of the synthesis step, e.g., as is the case in the SystemCoDesigner DSE methodology, an appropriate level of granularity can no longer be chosen in advance by the designer.

Monolithic clustering approaches [BL93, PBL95] for an automatic transformation of the level of granularity exist for data flow graphs consisting of purely static actors. Clustering replaces a connected subgraph of static actors by a composite actor implementing a schedule for the replaced actors derived at compile time by combining multiple static actors into one composite actor. Thus, clustering reduces the number of actors which have to be scheduled by the dynamic scheduler. This transformation reduces the scheduling overhead as it has been shown that compile-time scheduling of static actors produces more efficient implementations in terms of latency and throughput than run-time scheduling of the same actors by a dynamic scheduler. However, monolithic clustering ap-
proaches break down if even one non-static actor is present in the data flow graph. In contrast to this, for the first time, the second key contribution—the clustering methodology introduced in Chapter 4—can perform this transformation of the level of granularity for general data flow graphs. This goes significantly beyond previous approaches as it is not confined to monolithic clustering, but produces a Quasi-Static Schedule (QSS) for the generated composite actor. This enhances the power of the contribution in terms of avoiding deadlocks and, thus, increasing the number of actors that can be clustered. Hence, the clustering methodology is neither limited to data flow graphs consisting only of static actors, nor does it require a special treatment of these static actors by the developer of the SysteMoC application. For sufficiently fine-grained data flow graphs dominated by static actors, it is shown that a speedup by a factor of ten and beyond can be obtained by means of clustering.

Multiple challenges have been solved as part of this clustering methodology: (1) To identify the actors amenable to clustering, the SysteMoC language—in contrast to SystemC—enforces a distinction between communication and computation of an actor. This enables the design flow to determine the Model of Computation (MoC) used by a SysteMoC actor by means of classification [FHZT13*, ZFHT08*]. Actors that are classified as static data flow actors are amenable to the contributed clustering methodology. (2) A heuristic is contributed that maximizes the size of the created clusters and, thus, reduces the scheduling overhead for a single core software implementation. Determination of clusters by means of DSE for MPSoC implementations is also possible, but not explicitly presented as part of this thesis. (3) Two alternative clustering algorithms—an FSM-based and a rule-based representation of QSSs—are given that transform the level of granularity of a data flow graph by replacing islands of static actors in the data flow graph by composite actors implementing QSSs for the replaced actors. The rule-based representation trades latency and throughput in order to achieve a more compact code size for the QSS as compared to the Finite State Machine (FSM)-based representation. Intuitively, a QSS is a schedule in which a relatively large proportion of scheduling decisions have been made at compile time. A QSS executes sequences of statically scheduled actor firings at run time. These sequences have been determined at compile time by only postponing required scheduling decisions to run time. In contrast to the QSSs computed by the algorithm presented by Tripakis et al. [TBRL09, TBG+13], the algorithms employed by the contributed clustering algorithms will even generate retiming sequences in order to maximize the length of the sequences of statically scheduled actor firings and, thus, further reduce the scheduling overhead. (4) In order to extend the design space still further from data flow graphs with unbounded channel capacities to graphs with bounded channel capacities, a First In First Out (FIFO) channel capacity adjustment algorithm is given. The introduction of limited channel capacities into SysteMoC
models is necessary when refining a model for a subsequent implementation in hardware or software. Here, synthesis back-ends only support limited channels. Note that the determination of limited channel capacities that do not introduce deadlocks into a model is in general undecidable [Par95] for dynamic data flow graphs. However, the channel capacities for the FIFO channels connected to the cluster input and output ports cannot be used unmodified. The contributed adjustment algorithm ensures that clustering will not introduce an artificial deadlock into the clustered data flow graph if the original graph is sequence determinate. (5) Finally, a correctness proof is contributed for the introduced clustering methodology.

To conclude the presented design flow, the third key contribution of this work—the synthesis back-end introduced in Chapter 5—enables the transformation of executable specifications modeled in SysteMoC according to a set of design decisions determined via DSE into an implementation. The integration of this synthesis back-end with the SYSTEMCoDESIGNER DSE methodology allows the automatic optimization and generation of MPSoC systems—starting with an abstract actor-oriented model written in SysteMoC. Moreover, the presented synthesis back-end can be used to realize synthesis-in-the-loop for the SYSTEMCoDESIGNER DSE methodology. This enables SYSTEMCoDESIGNER to evaluate its design decisions based on real implementations, in contrast to the approximately-timed performance modeling performed via the Virtual Processing Components (VPC) [SFH+06] approach. Additionally, the implementations generated via the synthesis back-end can be employed to validate the annotated timing values [SGHT10] used for the VPC-based back annotation of design decisions into the SysteMoC application for the purpose of simulative evaluation at the ESL.

Examples for design decisions are the schedules generated by the clustering methodology, which are used by the synthesis back-end to reduce the scheduling overhead of the generated software, as well as the allocation and binding decisions. No manual refinement steps are needed, and so one major source of errors is eliminated. No manual optimizations are made. Instead, clustering is used to optimize the schedules for actors bound onto the same resource. In the generated implementations, the remaining actors which are not replaced by a composite actor as well as all composite actors are scheduled by the dynamic scheduler of each resource.

6.2 Future Research Directions

The presented work can be extended in several directions. From the DSE perspective, the decision of which actors to cluster can be delegated from the heuristics to the SYSTEMCoDESIGNER DSE methodology. In this way, both
architecture and clustering can be optimized together, thus, resulting in better optimized implementations. Moreover, while synthesis-in-the-loop can be used to evaluate these implementations, an analytic evaluation is preferable to speed up exploration. Here, simple heuristics like maximization of the length of the sequences of statically scheduled actor firings could be one objective for the DSE. A more advanced evaluation for clustering and binding decisions for islands of static actors can be performed by means of max-plus algebra to determine latency and throughput for the islands of static actors. In this way, adjustment of the FIFO channel capacities in order to maximize throughput for all FIFOs contained in the islands of static actors may also be considered.

From the clustering methodology perspective, an extension from clustering of islands of static actors to islands of sequence determinate, i.e., islands conforming to the Kahn Process Network (KPN) MoC, is an interesting challenge that would tremendously enlarge the design space for clustering. Here, the equivalence condition for sequence determinate clusters given in Chapter 4 can be reused to define safe cluster refinement operations for islands conforming to the KPN MoC. Open questions are how QSS should be represented for these kinds of clusters. If a finite state based representation is chosen, then what condition and assumptions for the KPN cluster are needed to guarantee the existence of a finite state based representation of its QSS. As a starting point for research in this direction, the work of Buck [Buc93] to schedule Boolean Data Flow (BDF) graphs may give interesting insights. Moreover, to guarantee a seamless design flow, classification of SysteMoC actors has to be extended to also recognize actors conforming to the KPN MoC.

From a synthesis perspective, code generation optimizations enabled by clustering are of interest. Here, techniques like buffer memory sharing of all FIFO channels inside the cluster or techniques like strength reductions of FIFO channels to local variable could be pursued. Moreover, binding actors not to one resource, but to a set of homogeneous resources might improve the possible scheduling flexibility of the dynamic scheduler. Here, techniques like work stealing [BL99] or the techniques presented for the multi-threaded simulation scheduler [HPB11] could be employed to compensate for the reduced scheduling flexibility induced by clustering actors together.
Eine Clustering-basierte Methode zur Abbildung von datenflussorientierten Anwendungen auf Ein-Chip-Multiprozessor-Systeme
Zusammenfassung

Durch neuartige Technologien und stetige Miniaturisierung werden zukünftige eingebettete Systeme eine immer höhere Rechenleistung – insbesondere durch die Integration mehrerer Rechenkerne – erzielen. Dies ermöglicht die Realisierung immer komplexerer Anwendungen. Gleichzeitig steigen die Anforderungen an die Effizienz solcher Systeme, insbesondere hinsichtlich ihres Energieverbrauchs. Hierfür ist es notwendig, eine speziell auf die Anwendung abgestimmte Architektur aus verschiedenen Rechenkernen (CPUs, DSPs, GPGPUs) und Hardware-Beschleunigern zu entwerfen bzw. Anwendungen auf gegebene Architekturen effizient abzubilden. Zukünftige Ansätze zum Entwurf solcher Systeme müssen deshalb die Aspekte Parallelität und Heterogenität inhärent und durchgehend von der Idee bis zur Realisierung unterstützen.

Modellierung:

Clustering:
Der Nachteil einer Modellierung auf Datenflussbasis ist die schwierige Generierung von effizienten Software-Implementierungen. Dieser Nachteil erwächst aus dem Missverhältnis zwischen dem sequentiellen imperativen Ausführungsmodell von Prozessoren und dem datengetriebenen Ausführungsmodell des Datenflussparadigmas. Im Bereich der Analyse wird mittels des in dieser Arbeit


Synthese:
Bibliography


[DTZ+08] Steven Derrien, Alexandru Turjan, Claudiu Zissulescu, Bart Kienhuis, and Ed F. Deprettere. Deriving efficient control in
Bibliography


Bibliography


Bibliography


[TBG+13] Stavros Tripakis, Dai N. Bui, Marc Geilen, Bert Rodiers, and Edward A. Lee. Compositionality in Synchronous Data Flow:


Bibliography

Author’s Own Publications


Author’s Own Publications


[ZHFT13*] Christian Zebelein, Christian Haubelt, Joachim Falk, and Jürgen Teich. Model-Based Representation of Schedules for Dataflow Graphs. In Christian Haubelt and Dirk Timmermann, editors,
Author’s Own Publications

### List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma$</td>
<td>The cluster operator</td>
<td>62</td>
</tr>
<tr>
<td>$\eta$</td>
<td>Firing count of an actor</td>
<td>13</td>
</tr>
<tr>
<td>$\eta^\text{rep}$</td>
<td>Repetition count of an actor</td>
<td>14</td>
</tr>
<tr>
<td>$\eta$</td>
<td>Vector of actor firings</td>
<td>131</td>
</tr>
<tr>
<td>$\eta^\text{rep}$</td>
<td>Repetition vector of a static DFG</td>
<td>14</td>
</tr>
<tr>
<td>$\eta^\text{rep}_\gamma$</td>
<td>Repetition vector of a subcluster $g_\gamma$</td>
<td>101</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>The empty signal</td>
<td>19</td>
</tr>
<tr>
<td>$\mu$</td>
<td>Phase of a CSDF actor</td>
<td>72</td>
</tr>
<tr>
<td>$\mu$</td>
<td>Classification candidate</td>
<td>72</td>
</tr>
<tr>
<td>$\nu$</td>
<td>A data value</td>
<td>8</td>
</tr>
<tr>
<td>$\mathcal{V}$</td>
<td>The universal set of data values</td>
<td>8</td>
</tr>
<tr>
<td>$\xi$</td>
<td>A cluster refinement operator</td>
<td>99</td>
</tr>
<tr>
<td>$\xi_{\text{SDF}}$</td>
<td>A cluster refinement operator producing SDF actors</td>
<td>101</td>
</tr>
<tr>
<td>$\xi_{\text{CSDF}}$</td>
<td>A cluster refinement operator producing CSDF actors</td>
<td>110</td>
</tr>
<tr>
<td>$\xi_{\text{FSM}}$</td>
<td>A cluster refinement operator producing an FSM-based QSS representation</td>
<td>136</td>
</tr>
<tr>
<td>$\xi_{\text{rules}}$</td>
<td>A cluster refinement operator producing a rule-based QSS representation</td>
<td>146</td>
</tr>
<tr>
<td>$\pi_N(x)$</td>
<td>Projection of a vector $x$ to a subset of its entries given by a subset $N \subseteq \mathcal{I}(x)$ of its index set</td>
<td>131</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Sequence of rule applications, e.g., $\rho = \langle r_1, r_2, \ldots, r_n \rangle$</td>
<td>147</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Number of phases of a CSDF actor</td>
<td>15</td>
</tr>
<tr>
<td>$\omega$</td>
<td>Annotation tuple used in actor classification</td>
<td>75</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>The empty set</td>
<td>62</td>
</tr>
<tr>
<td>$\mathbb{N}$</td>
<td>The set of natural numbers</td>
<td>12</td>
</tr>
<tr>
<td>$\mathbb{N}_0$</td>
<td>The set of non-negative integers</td>
<td>8</td>
</tr>
<tr>
<td>$\mathbb{N}_{0,\infty}$</td>
<td>The set of non-negative integers including infinity</td>
<td>17</td>
</tr>
<tr>
<td>$\mathbb{Z}$</td>
<td>The set of integers</td>
<td>77</td>
</tr>
<tr>
<td>$</td>
<td>X</td>
<td>$</td>
</tr>
<tr>
<td>$X^*$</td>
<td>Finite Kleene closure of a value set $X$</td>
<td>8</td>
</tr>
<tr>
<td>$X^{**}$</td>
<td>Infinite Kleene closure of a value set $X$</td>
<td>17</td>
</tr>
</tbody>
</table>
### List of Symbols

- $\langle x_1, x_2, x_3 \rangle$: Sequence ................................................................. 10
- $\sqsubset$: Prefix order for signals .................................................. 20
- $\#$: The sequence length operator .................................................. 17
- $\#_x x$: Number of occurrences of a value $x$ in a sequence $x$ ........ 163
- $\triangleright$: The sequence concatenation operator ............................. 18
- $f \circ g$: Function composition operator, i.e., $f \circ g(x) \equiv f(g(x))$ .... 59
- $a$: Actor in a DFG ................................................................. 8
- $a_{in}$: An input actor of a cluster .................................................. 62
- $a_{out}$: An output actor of a cluster .............................................. 62
- $a_\gamma$: Composite actor derived from cluster $g_\gamma$ via refinement ... 88
- $a$: Sequence of actor firings, e.g., $a = \langle a_1, a_5, a_4, a_6, a_3, a_2 \rangle$ .... 10
- $\text{adj}_{ii}$: Adjustment of the cluster input channel capacities to handle input-to-input back pressure .................. 181
- $\text{adj}_{i2o}$: Adjustment of the cluster input channel capacities to handle input-to-output back pressure ............ 177
- $\text{adj}_{o2o}$: Adjustment of the cluster output channel capacities to handle output-to-output back pressure .......... 183
- $A$: Set of actors ........................................................................... 8
- $A_{EC}$: An equivalence class, i.e., a set of actors of a cluster candidate that share the same set of predecessor input actors and successor output actors .......... 125
- $A_I$: The set of input actors of a cluster ........................................... 62
- $A_O$: The set of output actors of a cluster ........................................ 62
- $A_S$: The set of static actors of a graph .......................................... 124
- $c$: Channel in a DFG ................................................................. 8
- $\text{clash}$: A symmetric relation between equivalence classes that evaluates to true if the actors of these equivalence classes must not be contained in the same cluster ............... 125
- $\text{conflict}$: Conflict relation for two rules .......................................... 149
- $\text{cons}(c)$: Consumption rate on channel $c$ ...................................... 9
- $\text{cons}(i)$: Consumption rate on input port $i$ .................................... 22
- $\text{cons}(t, i)$: Consumption rate of transition $t$ on input port/channel $i$ .... 48
- $\text{cons}(a_1, a_2)$: Consumption rate of actor $a_2$ on the channel between the actors $a_1$ and $a_2$ ....................... 111
- $C$: Set of channels ................................................................. 8
- $\text{delay}$: The delay function .......................................................... 8
- $\text{dep}$: The input/output dependency function .................................. 137
- $e$: An edge ................................................................................. 25
- $E$: A set of edges (usually partitioned into more specific subsets) ....... 25
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>A function operating on finite signals</td>
</tr>
<tr>
<td>$f_{\text{action}}$</td>
<td>An action function</td>
</tr>
<tr>
<td>$f_{\text{guard}}$</td>
<td>A guard function</td>
</tr>
<tr>
<td>$f$</td>
<td>The Boolean value false</td>
</tr>
<tr>
<td>$\text{firings}$</td>
<td>Mapping function from a cluster state to the number of firings of all actors in the cluster</td>
</tr>
<tr>
<td>$\mathcal{F}$</td>
<td>The set of all functions operating on finite signals</td>
</tr>
<tr>
<td>$\mathcal{F}_{\text{action}}$</td>
<td>A set of SysteMoC actions</td>
</tr>
<tr>
<td>$\mathcal{F}_{\text{guard}}$</td>
<td>A set of SysteMoC guards</td>
</tr>
<tr>
<td>$g$</td>
<td>A data flow graph</td>
</tr>
<tr>
<td>$g_a$</td>
<td>An architecture graph</td>
</tr>
<tr>
<td>$g$</td>
<td>A cluster or network graph</td>
</tr>
<tr>
<td>$g\gamma$</td>
<td>A subcluster of the cluster or network graph $g$</td>
</tr>
<tr>
<td>$g_n$</td>
<td>A network graph</td>
</tr>
<tr>
<td>$G$</td>
<td>A set of data flow graphs or clusters</td>
</tr>
<tr>
<td>$G_n$</td>
<td>A set of data flow graphs</td>
</tr>
<tr>
<td>$H_{\text{dep}}(a_{\text{out}})$</td>
<td>The set of input/output dependency function values for an output actor</td>
</tr>
<tr>
<td>$L$</td>
<td>A set of communication links between resources</td>
</tr>
<tr>
<td>$M$</td>
<td>A set of mapping edges</td>
</tr>
<tr>
<td>$\text{interleave}$</td>
<td>Function deriving a set of cluster states by interleaving actor firings from a given set of states</td>
</tr>
<tr>
<td>$\text{io}$</td>
<td>Mapping function from a cluster state to the number of consumed (negative) and produced (positive) tokens on the cluster ports</td>
</tr>
<tr>
<td>$I$</td>
<td>A set of input ports</td>
</tr>
<tr>
<td>$\mathcal{I}$</td>
<td>Function returning the index set of a vector or sequence</td>
</tr>
<tr>
<td>$k$</td>
<td>A Boolean function used as a transition guard</td>
</tr>
<tr>
<td>$k^{\text{io}}$</td>
<td>A Boolean function used as part of a transition guard to check the availability of tokens and free places</td>
</tr>
<tr>
<td>$\mathcal{K}$</td>
<td>A Kahn function</td>
</tr>
<tr>
<td>$l$</td>
<td>Lower bound for a rule</td>
</tr>
<tr>
<td>$\text{lfp}(x = f(x))$</td>
<td>The minimal value $x$ solving the equation $x = f(x)$</td>
</tr>
<tr>
<td>$\text{max } X$</td>
<td>Maximal value $x \in X$ of a set $X$</td>
</tr>
<tr>
<td>$\text{max } X$</td>
<td>Least upper bound of a set $X$ of partially ordered vectors</td>
</tr>
<tr>
<td>$\text{min } X$</td>
<td>Minimal value $x \in X$ of a set $X$</td>
</tr>
<tr>
<td>$\text{mod}$</td>
<td>Modulo operation for integers</td>
</tr>
<tr>
<td>$q \mod \eta$</td>
<td>Modulo operation for cluster states</td>
</tr>
<tr>
<td>$r \mod \eta$</td>
<td>Modulo operation for rules</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>$O$</td>
<td>A set of output ports ........................................ 43</td>
</tr>
<tr>
<td>prod$(c)$</td>
<td>Production rate on channel $c$ ........................................ 9</td>
</tr>
<tr>
<td>prod$(o)$</td>
<td>Production rate on output port $o$ ............................... 22</td>
</tr>
<tr>
<td>prod$(t,o)$</td>
<td>Production rate of transition $t$ on output port/channel $o$ .... 48</td>
</tr>
<tr>
<td>prod$(a_1,a_2)$</td>
<td>Production rate of actors $a_1$ and $a_2$ ......................... 111</td>
</tr>
<tr>
<td>$P$</td>
<td>A set of ports (usually partitioned into a set of input and output ports) ........................................ 44</td>
</tr>
<tr>
<td>$q$</td>
<td>A state ........................................................................ 25</td>
</tr>
<tr>
<td>$q_0$</td>
<td>An initial state .................................................. 25</td>
</tr>
<tr>
<td>$q_{src}$</td>
<td>Source state of a transition .................................. 25</td>
</tr>
<tr>
<td>$q_{dst}$</td>
<td>Destination state of a transition ................................. 25</td>
</tr>
<tr>
<td>$q$</td>
<td>A state of a cluster ............................................... 130</td>
</tr>
<tr>
<td>$q_0$</td>
<td>An initial state of a cluster .................................. 130</td>
</tr>
<tr>
<td>$q_{src}$</td>
<td>Source state of a cluster FSM transition ................... 132</td>
</tr>
<tr>
<td>$q_{dst}$</td>
<td>Destination state of a cluster FSM transition ................ 132</td>
</tr>
<tr>
<td>$Q$</td>
<td>A set of states .................................................. 133</td>
</tr>
<tr>
<td>$Q$</td>
<td>A set of cluster states ........................................ 133</td>
</tr>
<tr>
<td>$Q_\gamma$</td>
<td>State space of the cluster $g_\gamma$ .......................... 133</td>
</tr>
<tr>
<td>$Q^{dyn}_\gamma$</td>
<td>State space of the cluster $g_\gamma$ if dynamically scheduled .... 177</td>
</tr>
<tr>
<td>$Q_{ini}$</td>
<td>The set of initial input/output dependency states ........... 140</td>
</tr>
<tr>
<td>$Q_{lfp}$</td>
<td>The set of interleaved states ................................... 140</td>
</tr>
<tr>
<td>$r$</td>
<td>Resource in an architecture graph ............................. 37</td>
</tr>
<tr>
<td>$R$</td>
<td>A set of resources .............................................. 206</td>
</tr>
<tr>
<td>$r$</td>
<td>A rule ...................................................................... 146</td>
</tr>
<tr>
<td>redundant</td>
<td>Function selecting all redundant rules of a set of rules ........ 153</td>
</tr>
<tr>
<td>resolve</td>
<td>Function performing one iteration of conflict resolution .. 152</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>A finite state machine ........................................... 25</td>
</tr>
<tr>
<td>$R$</td>
<td>A set of rules .................................................. 146</td>
</tr>
<tr>
<td>$R_{fin}$</td>
<td>The set of final rules ........................................... 146</td>
</tr>
<tr>
<td>$R_{ini}$</td>
<td>The set of initial input/output dependency rules .......... 147</td>
</tr>
<tr>
<td>$R_{lfp}$</td>
<td>The set of conflict-resolved rules ................................ 147</td>
</tr>
<tr>
<td>$R_{red}$</td>
<td>The set of redundant rules ...................................... 153</td>
</tr>
<tr>
<td>$s$</td>
<td>A signal .......................................................... 17</td>
</tr>
<tr>
<td>$s$</td>
<td>Tuple of signals .................................................. 18</td>
</tr>
<tr>
<td>size</td>
<td>The channel capacity function .................................. 49</td>
</tr>
<tr>
<td>snk$(c)$</td>
<td>Function returning the sink actor of a channel $c$ .............. 8</td>
</tr>
<tr>
<td>src$(c)$</td>
<td>Function returning the source actor of a channel $c$ .......... 8</td>
</tr>
<tr>
<td>state</td>
<td>Function deriving the cluster state from the signals consumed and produced by the cluster ...................... 133</td>
</tr>
</tbody>
</table>
Function deriving a cluster state from a given number of input actor firings.

**state**<sub>dep</sub> ................................................................. 139

* S  The set of all signals .................................................. 17

* S  The set of all vectors of signals .................................... 18

* t  A transition ....................................................................... 25

* t<sub>EXEC</sub>  Overall execution time .................................... 88

* t<sub>SA</sub>  Overall scheduling time ....................................... 88

* t<sub>TA</sub>  Overall computation time ..................................... 88

* t  The Boolean value true ..................................................... 23

* tail  The tail function ........................................................... 18

* T  A set of transitions ........................................................... 25

* TO<sub>x</sub>  The set of tightly ordered pairs for a set of states \( Q_x \) .... 143

* u  Upper bound for a rule ..................................................... 146

* v  A vertex ............................................................................ 25

* V  A set of vertices (usually partitioned into more specific subsets) ........................................................................ 25
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APG</td>
<td>Acyclic Precedence Graph</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>BB</td>
<td>Basic Block</td>
</tr>
<tr>
<td>BDF</td>
<td>Boolean Data Flow</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block Random Access Memory</td>
</tr>
<tr>
<td>CAL</td>
<td>Cal Actor Language</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>CFSM</td>
<td>Codesign Finite State Machine</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>clang</td>
<td>C Language Family Frontend of the LLVM Compiler Infrastructure</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSDF</td>
<td>Cyclo-Static Data Flow</td>
</tr>
<tr>
<td>CT</td>
<td>Continuous Time</td>
</tr>
<tr>
<td>DDF</td>
<td>Dennis Data Flow</td>
</tr>
<tr>
<td>DE</td>
<td>Discrete Event</td>
</tr>
<tr>
<td>DFG</td>
<td>Data Flow Graph</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>DSSF</td>
<td>Deterministic SDF with shared FIFOs</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>EDK</td>
<td>Xilinx Embedded Development Kit</td>
</tr>
<tr>
<td>ESL</td>
<td>Electronic System Level</td>
</tr>
<tr>
<td>FF</td>
<td>Flip-Flop</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
</tbody>
</table>
### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunState</td>
<td>Functions Driven by State Machines</td>
</tr>
<tr>
<td>GALS</td>
<td>Globally Asynchronous Locally Synchronous</td>
</tr>
<tr>
<td>HDF</td>
<td>Heterochronous Data Flow</td>
</tr>
<tr>
<td>HMUL</td>
<td>Hardware Multiplier</td>
</tr>
<tr>
<td>HSDF</td>
<td>Homogeneous (Synchronous) Data Flow</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>KPN</td>
<td>Kahn Process Network</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MOEA</td>
<td>Multi-Objective Evolutionary Algorithm</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multi-Processor System-on-Chip</td>
</tr>
<tr>
<td>NDF</td>
<td>Non-Determinate Data Flow</td>
</tr>
<tr>
<td>OSCI</td>
<td>Open SystemC Initiative</td>
</tr>
<tr>
<td>PGAN</td>
<td>Pairwise Grouping of Adjacent Nodes</td>
</tr>
<tr>
<td>PSOS</td>
<td>Periodic Static-Order Schedule</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SAS</td>
<td>Single Appearance Schedule</td>
</tr>
<tr>
<td>SAT</td>
<td>Boolean Satisfiability</td>
</tr>
<tr>
<td>SCC</td>
<td>Strongly Connected Component</td>
</tr>
<tr>
<td>SPDF</td>
<td>Synchronous Piggybacked Data Flow</td>
</tr>
<tr>
<td>SR</td>
<td>Synchronous Reactive</td>
</tr>
<tr>
<td>QSS</td>
<td>Quasi-Static Schedule</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTTI</td>
<td>Run Time Type Information</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SysteMoC</td>
<td>SystemC Models of Computation</td>
</tr>
<tr>
<td>Verilog</td>
<td>Verilog Hardware Description Language</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware Description Language</td>
</tr>
<tr>
<td>VPC</td>
<td>Virtual Processing Components</td>
</tr>
</tbody>
</table>
Index

actor, 7
actor FSM, 46
deadlock-free SDF graph, 13
functionality state, 45
FunState, 25

consistent Synchronous Data Flow (SDF)
graph, 13
coalescing of actor firings, 104
coelescing operation, 106
communication behavior, 10, 44, 52
composite actor, 23, 90
conflict rules, 149
conflict-resolved rules, 153
grouping of actor firings, 111

bounded memory execution, 10
boolean data flow, 21
channel, 7
delay, 7
delay-less cycle, 10
dynamic data flow, 17
dynamic scheduler, 85
dynamic scheduling, 85

channel capacity, 7, 48
classification candidate, 72
closed system, 61
classifier, 59
classifier FSM, 89, 90
classifier candidate, 124
classifier environment, 62
classifier input channel, 173
classifier output channel, 173
classification partitioning heuristic, 123
classifier state, 130
classifier state space, 130
configuration, 90
clustering condition, 121
conflict rules, 149
conflict-resolved rules, 153
deadlock-free SDF graph, 13
delay, 7
delay-less cycle, 10
delay-less directed path, 112
dynamic data flow, 17
dynamic scheduler, 85
dynamic scheduling, 85
equivalence class, 127
equivalent behavior in data flow, 97

local state machine, 27
fireable, 7
firing, 7
functionality guard, 47
functionality state, 45

Index
heterochronous data flow, 30
hierarchical network graph, 62
homogeneous data flow, 10

implementation graph, 37
incoming transition, 46
initial cluster state, 90, 131
initial input/output dependency states, 140
initial token, 7
input actor, 62, 63, 107
input guard, 47
input ports, 19
input predicate, 47
input/output dependency function, 137
input/output dependency function value, 136
intermediate actor, 62, 131
interval boundary vectors, 146
iteration, 11

Kahn process network, 19

leaf states, 57
length of a signal, 17
length of an iteration, 11

marked graph, 10

network graph, 48
non-hierarchical network graph, 48

open system, 61
outgoing transition, 46
output actor, 62, 63, 107
output guard, 47
output ports, 19
output predicate, 47
overall computation time, 88
overall execution time, 42, 89
overall overhead, 93
overall scheduling time, 88
parallel composition, 59
partial repetition vector, 84, 146
periodic static-order schedule, 10, 11
ports, 19
problem graph, 37, 205
production rate, 9
production rate function, 12, 15, 46, 48, 111
quasi-static schedule, 89
reachable cluster state, 162
redundant rules, 153
repetition vector sum, 145
resource, 37
retiming, 108
rule, 146
rule-based clustering, 145
safe cluster refinement, 100
scheduling overhead, 88
SDF to marked graph unfolding, 103
sequence determinate, 24
sequence of actor firings, 11
set of initial input/output dependency rules, 147
set of input/output dependency function values, 136
signal, 17
source state of a transition, 46
specification graph, 37, 207
static cluster, 83
static data flow, 9
static data flow actor, 10
subcluster, 61
synchronous data flow, 12
synchronous reactive domain, 59
synthesis back-end, 203
synthesis-in-the-loop, 203
SystemC, 34
SysteMoC, 44
token, 7
transition contraction condition, 73
unsafe cluster refinement, 100
worst case cluster environment, 99
xor states, 57
Joachim Falk was born in 1977 and grew up around Erlangen, Germany. He earned his university admission by visiting the technical school (Fachoberschule) in Erlangen and started his study of electrical engineering in 1997 at the Georg Simon Ohm University of Applied Sciences in Nuremberg, Germany. There, he earned his degree in Electrical Engineering with a Specialization in Data Processing in 2002. In 2003, he joined the chair of Hardware/Software Co-Design at the Friedrich-Alexander-Universität Erlangen-Nürnberg under the supervision of Prof. Dr.-Ing. Jürgen Teich. From 2006 onward, he has been working as a researcher in the project “Actor-Oriented Synthesis and Optimization of Digital Hardware/Software-Systems at the Electronic System Level,” which is funded by the German Research Foundation (DFG). Joachim Falk has been a reviewer for several international journals. His main research focuses are data flow languages and transformations for data flow graphs in the context of embedded system design.