A Model-Based Approach for the Specification and Refinement of Streaming Applications

Dissertation
zur
Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

Rostock, 2014

vorgelegt von

Christian Zebelein
geboren am 26.01.1983 in Erlangen

urn:nbn:de:gbv:28-diss2015-0075-9

Gutachter:

- **Prof. Dr.-Ing. habil. Christian Haubelt** (Universität Rostock, Fakultät für Informatik und Elektrotechnik, Institut für Angewandte Mikroelektronik und Datentechnik)
- **Prof. Dr.-Ing. Jürgen Teich** (Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 12 (Hardware-Software-Co-Design))
- **Prof.dr.ir. Twan Basten** (Eindhoven University of Technology, Department of Electrical Engineering)

Datum der Einreichung: 27.04.2014
Datum der Verteidigung: 22.09.2014

Abstract

Today, embedded systems can be found in a wide range of applications like transportation systems, consumer electronics, medical equipment, industrial applications, or computer networking devices. Embedded systems are hardware/software systems which are dedicated to a particular task in the context of a larger system. Depending on the application, embedded systems have to meet a wide range of constraints like performance, power, area, reliability, safety, or security constraints. Besides these constraints, the time to market becomes more and more important. It follows that designing and programming embedded systems is a challenging task.

A design methodology is therefore required which supports the automatic decision-making and refinement process at system level, as manually performing these tasks is time-consuming and error-prone. Here, model-based design flows could be a solution. Supporting the automatic decision-making process at system level, a plethora of analysis and optimization techniques have been developed for many different formal models. In contrast, the automatic refinement of the application model in a model-based design flow is still an open problem.

This thesis therefore proposes novel approaches for the specification and refinement of streaming applications in a model-based design flow. To this end, it focuses on dataflow models. In a dataflow model, concurrent modules communicate and synchronize via packets transmitted over channels. Dataflow models can be used to implement a wide range of streaming applications as commonly found in the multimedia or networking domain. The most important contributions of this thesis can be summarized as follows:

1. The proposed dataflow model provides for the most general dataflow semantics, namely dynamic dataflow (DDF), which allows a wide range of streaming applications to be implemented in principle.

2. Despite its expressiveness, the proposed dataflow model still permits the identification of restricted dataflow models. In turn, this enables the application of domain-specific analysis and optimization techniques that have been developed for these restricted dataflow models.

3. At system level, binding and scheduling decisions are incorporated into the dataflow model by means of hierarchical modules. In contrast to existing

hierarchical dataflow models, hierarchical modules have the same dataflow semantics as non-hierarchical modules, which are used to implement the behavior of the application. In turn, the same analysis and optimization techniques can be applied to hierarchical and non-hierarchical modules.

4. In the proposed seamless model-based design flow, the hierarchical dataflow model at system level constitutes the input model for subsequent hardware/software synthesis steps at the next lower levels of abstraction. Thus, a wide range of scheduling strategies can be synthesized in principle. More importantly, complex model-based optimizations can still be applied during synthesis at these lower levels of abstraction. As these optimizations can be automatically performed based on the proposed dataflow model, the overall modeling complexity is greatly reduced. Moreover, the proposed model-based optimizations considerably extend the design space, as different configurations can be automatically synthesized and evaluated during design space exploration in principle.

Kurzfassung

Heutzutage sind eingebettete Systeme in vielfältigen Anwendungsbereichen anzutreffen. Beispiele hierfür sind Verkehrsmittel, Unterhaltungs- und Haushaltselektronik, medizinische Geräte, industrielle Anwendungen, sowie Computernetzwerke. Eingebettete Systeme sind Hardware/Software-Systeme, die im Rahmen eines größeren Systems eine bestimmte Aufgabe erfüllen. Je nach Anwendung müssen eingebette Systeme unterschiedlichste Rahmenbedingungen einhalten. Beispiele hierfür sind Rechenleistung, Stromverbrauch, Größe, Zuverlässigkeit, Betriebssicherheit oder Informationssicherheit. Neben diesen Rahmenbedingungen spielt auch die Produkteinführungszeit (engl. "time to market") eine immer wichtigere Rolle. Somit stellt der Entwurf und die Programmierung von eingebetteten Systemen eine schwierige Aufgabe dar.

Aus diesem Grund wird eine Entwurfsmethodik benötigt, die es erlaubt, automatisch Entscheidungen auf Systemebene zu treffen, sowie die getroffenen Entscheidungen automatisch in der Anwendung umzusetzen. Hier könnten modellbasierte Entwurfsflüsse eine Lösung darstellen. In einem modellbasierten Entwurfsfluss dient ein formales Anwendungsmodell als Eingabe für die Entscheidungsfindung und Verfeinerung auf Systemebene. Um die automatische Entscheidungsfindung auf Systemebene zu unterstützen, wurden bereits eine Vielzahl von Analyse- und Optimierungsverfahren für unterschiedlichste formale Modelle entwickelt. Andererseits ist die automatische Verfeinerung des Anwendungsmodells in einem modellbasierten Entwurfsfluss noch ein ungelöstes Problem.

In dieser Arbeit werden deshalb neuartige Ansätze zur Spezifikation und Verfeinerung von Streaming-Anwendungen in einem modellbasierten Entwurfsfluss vorgestellt. Dazu konzentriert sich die Arbeit auf Datenflussmodelle. In einem Datenflussmodell kommunizieren nebenläufige Module durch Pakete miteinander, die über Kanäle weitergeleitet werden. Datenflussmodelle eignen sich gut zur Modellierung von Streaming-Anwendungen, die häufig im Multimedia- oder Netzwerkbereich vorzufinden sind. Die wichtigsten Beiträge dieser Arbeit lassen sich wie folgt zusammenfassen:

1. Das vorgeschlagene Datenflussmodell besitzt die allgemeinste Datenflusssemantik, nämlich dynamischen Datenfluss, und ermöglicht somit die Implementierung einer Vielzahl von Streaming-Anwendungen.

2. Trotz seiner Ausdruckskraft erlaubt das vorgeschlagene Datenflussmodel dennoch die Erkennung von eingeschränkten Datenflussmodellen. Dies wiederum ermöglicht den Einsatz von domänenspezifischen Analyse- und Optimierungsverfahren, die für diese eingeschränkten Datenflussmodelle entwickelt worden sind.

3. Auf Systemebene werden getroffene Bindungs- und Ablaufplanungsentscheidungen durch hierarchische Module in der Anwendung umgesetzt. Im Gegensatz zu anderen hierarchischen Datenflussmodellen besitzen die hierarchischen Module dieselbe Datenflusssemantik wie die nicht-hierarchischen Module, die verwendet werden, um das Anwendungsverhalten zu implementieren. Dies wiederum ermöglicht den Einsatz derselben Analyse- und Optimierungsverfahren sowohl für hierarchische als auch für nicht-hierarchische Module.

4. Im vorgestellten durchgängig modellbasierten Entwurfsfluss stellt das hierarchische Datenflussmodell auf Systemebene das Eingabemodel für anschließende Syntheseschritte auf den nächstniedrigeren Abstraktionsebenen dar. Somit lassen sich prinzipiell vielfältige Ablaufplanungsverfahren synthetisieren. Wichtiger jedoch ist die Tatsache, dass komplexe modellbasierte Optimierungsverfahren somit auch noch während der Synthese auf diesen niedrigeren Abstraktionsebenen eingesetzt werden können. Da diese Optimierungsverfahren basierend auf dem vorgeschlagenen Datenflussmodell automatisch angewendet werden können, wird damit der Modellierungsaufwand erheblich reduziert. Darüber hinaus erweitern die vorgeschlagenen Optimierungsverfahren den Entwurfsraum deutlich, da verschiedene Konfigurationen während der Entwurfsraumexploration prinzipiell automatisch synthetisiert und evaluiert werden können.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to professor Christian Haubelt and professor Jürgen Teich. As advisors, their continued support and guidance over the past few years gave me invaluable inspiration for my work and this thesis.

Special thanks go to my colleagues from the University of Erlangen-Nuremberg and the University of Rostock. In particular, I want to thank Joachim Falk, Jens Gladigau, Martin Streubühr, Daniel Ziener, and Lars Middendorf, as they helped me to develop and implement solutions for various problems. I would also like to thank Rainer Dorsch, who made it possible for me to contribute some of my ideas to a successful industrial collaboration over the course of several years.

Last but not least, this thesis would not have been possible without the support of my family and close friends, who allowed me to take my mind off of my work and this thesis during my much-too-infrequent visits in the past few years. You know who you are!

Contents

1 Introduction

Today, embedded systems can be found in a wide range of applications. Well-known examples are transportation systems like automobiles or airplanes, consumer electronics like smartphones, digital cameras, washing machines, or home automation systems, medical equipment for the monitoring of vital signs or medical imaging, industrial applications like factory automation systems, or computer networking devices like routers or network adapters.

Embedded systems are hardware/software systems which are dedicated to a particular task in the context of a larger system. Typically, they do not interface with the environment by means of familiar human interface devices like keyboards, mice, or computer monitors. Instead, serial or network connections are typically used by embedded systems to exchange data with the environment. Additionally, sensors may be used to gather information from the environment, while actuators may be used to influence the environment.

Depending on the application, embedded systems have to meet a wide range of constraints. *Performance constraints* like latency or throughput, as well as *power and area constraints* can often be found in consumer electronics or networking devices. *Reliability* is most important for mission-critical embedded systems like those employed in airplanes or spacecraft, while automation systems are expected to operate continuously without user interaction for extended time spans like years or even decades. *Safety constraints* can be found in numerous applications where failure may lead to the harming of humans. Here, medical equipment or transportation devices are well-known examples. In particular, safety-critical applications are typically subject to real-time constraints which guarantee that the system reacts to certain events within strict time constraints known as deadlines. Here, air bag control systems and anti-lock braking systems (ABS) employed in automobiles are prominent examples. *Security* is important for devices which may handle confidential data which must not be accessed or modified by unauthorized users. Here, cryptographical algorithms are typically used to protect information.

Besides these constraints, the *time to market* becomes more and more important, especially in the context of highly competitive global markets. Here, new features must be conceived, designed, and implemented before similar features

are integrated into products from competitors or the product demand is falling. However, the time to market should not be reduced at the expense of the relevant constraints outlined above. Thus, designing and programming embedded systems is a challenging task:

1. At the system level, a wide range of heterogeneous resources are at the designer's disposal in order to meet the constraints identified for the particular task to be performed by an embedded system: *Computing resources* like microprocessors, dedicated hardware accelerators, or analog and mixed-signal components can be used to execute certain parts of the application. For example, a dedicated hardware accelerator may be used instead of a general-purpose microprocessor in order to improve the performance of the final product. *Storage resources* like random-access memory (RAM) or flash memory are used to store data. Here, storage size, access times and the interface bandwidth are important factors w.r.t. the overall design constraints. Finally, *communication resources* like buses or crossbars provide for the communication between computing and storage resources. Thus, some questions to be answered by the designer at system level can be stated as follows: Which resources should be selected from this set of possible resources? Which parts of the application should be bound to which resources that have been selected? How should resource contention be resolved if multiple parts are bound to the same resource? Without a design flow supporting this decision-making process, the designer must answer these questions in an ad-hoc manner, which almost certainly would result in some of the identified constraints not being met, thereby increasing the time to market. Thus, a design methodology at system level is required which supports the automatic decision-making process at system level.

2. Subsequently, the made decisions must be incorporated into the application such that the refined application ties in with the design methodologies used at the next lower levels of abstraction in order to provide for a seamless design flow. However, if done manually, this refinement step is time-consuming and error-prone. Thus, the design methodology at system level should also support the automatic refinement process at system level.

In order to solve these problems, *model-based* design flows could be a solution. In a model-based design flow, a formal application model serves as input to the decision-making and refinement process at system level. In turn, this permits the application of analysis and optimization techniques that have been developed for the formal model in question, thereby supporting the automatic decision-making process at system level. While a plethora of analysis and optimization techniques have been developed for many different formal models, the automatic refinement of the application model in a model-based design flow is still an open problem.

This thesis therefore proposes some novel approaches for the specification and refinement of streaming applications in a model-based design flow. To this end, it focuses on *dataflow models*. In a dataflow model, *concurrent modules* communicate and synchronize via *packets* transmitted over *channels*. Dataflow models can be used to implement a wide range of streaming applications as commonly found in the multimedia or networking domain. Examples are video compression algorithms or the packet processing performed by various components in computer networks. The most important contributions of this thesis can be summarized as follows:

1. The proposed dataflow model provides for the most general dataflow semantics, namely *dynamic dataflow* (DDF), which allows a wide range of streaming applications to be implemented in principle.

2. Despite its expressiveness, the proposed dataflow model still permits *the identification of less expressive dataflow models*. In turn, this enables the application of domain-specific analysis and optimization techniques that have been developed for these less expressive dataflow models.

3. At system level, binding and scheduling decisions are incorporated into the dataflow model by means of *hierarchical modules*. In contrast to existing hierarchical dataflow models, hierarchical modules have the same dataflow semantics as non-hierarchical modules, which are used to implement the behavior of the application. In turn, the same analysis and optimization techniques can be applied to hierarchical and non-hierarchical modules.

4. In the proposed seamless model-based design flow, the hierarchical dataflow model at system level constitutes the input model to subsequent synthesis steps at the next lower levels of abstraction. This enables *complex model-based optimizations* to be applied even at these lower levels of abstraction.

In the following, these contributions are discussed in more detail by means of the structure of this thesis (cf. Figure 1.1).

1.1 Summary of Contributions

Large parts of the solutions and results presented in this thesis have been published previously. A list of all publications authored and co-authored by the author of this thesis can be found starting at page 183. The thesis at hand unifies and expands on several aspects of these publications in order to provide a consistent presentation from the formal input model at system level to the refined model at lower levels of abstraction, as described in the following.

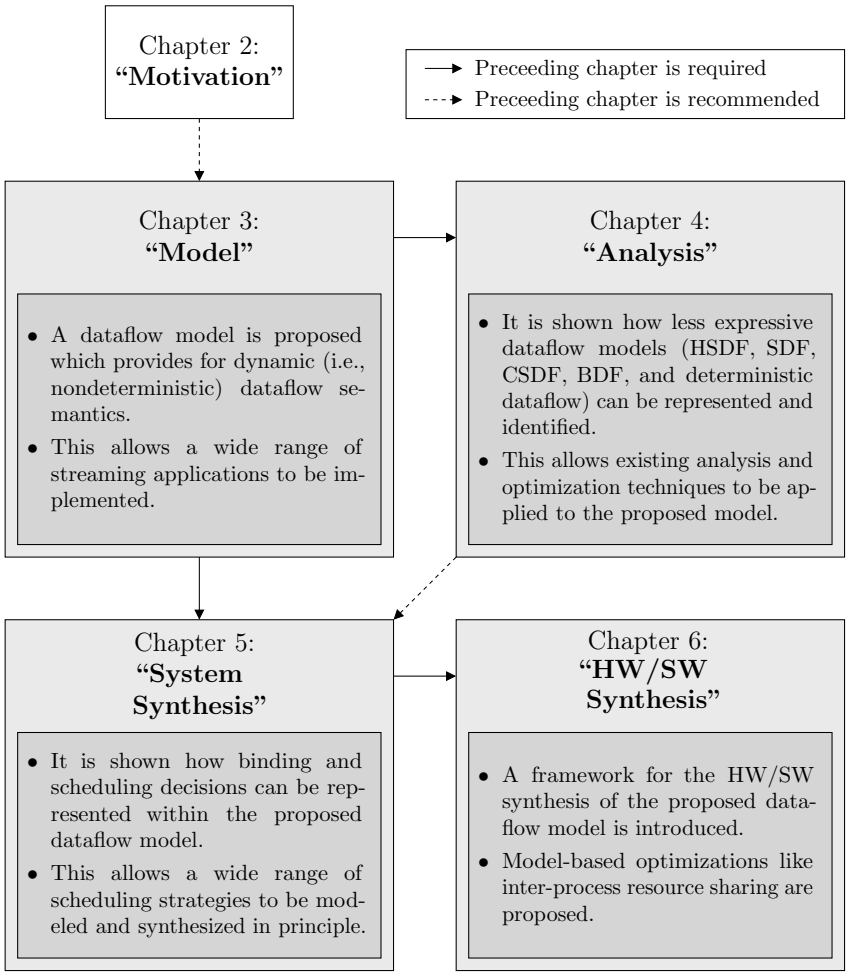


Figure 1.1: Dependencies between the main chapters of this thesis. While all chapters contain own contributions, the most important contributions can be found within the shaded chapters.

Dynamic Dataflow Model

This thesis proposes a *seamless model-based design flow* from the specification at system level to lower levels of abstraction. To this end, it focuses on *dataflow models*. In a dataflow model, *concurrent modules* communicate and synchronize via *packets* transmitted over *channels*. Dataflow models can be used to implement a wide range of streaming applications as commonly found in the multimedia or networking domain. The non-hierarchical dataflow model has been presented in [ZHF+13; ZHFT13]. A unified and comprehensive description of the proposed dataflow model is given in Chapter 3 (cf. Figure 1.1). Suffice to say, the proposed dataflow model provides for the most general dataflow semantics, namely DDF, which allows a wide range of streaming applications to be implemented in

principle. In this context, the extension of the model by hierarchical finite state machines (FSMs) (which are not to be confused with hierarchical modules) has been proposed in [ZFH+10]. Please note that Chapter 2 reviews the typical steps in the design of digital systems from the specification to an implementation in more detail, and additionally motivates the proposed dataflow model by means of a JPEG decoder.

Representation and Identification of Restricted Dataflow Models

The proposed dataflow model provides for the most general dataflow semantics, namely DDF. Nevertheless, in contrast to existing dataflow models, the proposed dataflow model still permits the identification of less expressive dataflow models. In turn, this enables the application of domain-specific analysis and optimization techniques that have been developed for these less expressive dataflow models. Chapter 4 discusses the representation and identification of some well-known less expressive dataflow models (cf. Figure 1.1). The identification of static dataflow models has been proposed in [ZFHT08]. The corresponding discussion in Chapter 4 has been adapted to the proposed dataflow model, and some proofs have been added. The identification of more expressive dataflow models has been partially adapted from [Lee97; BHLP09]. The representation of all considered less expressive dataflow models by the proposed dataflow model represents novel work.

Model-Based Representation of Binding and Scheduling Decisions

The refinement step at system level requires binding and scheduling decisions to be incorporated into the dataflow model. To this end, *hierarchical modules* are used. In contrast to existing hierarchical dataflow models, hierarchical modules have the same dataflow semantics as non-hierarchical modules, which are used to implement the behavior of the application. In turn, the same analysis and optimization techniques can be applied to hierarchical and non-hierarchical modules. The hierarchical model has been proposed in [ZHF+13; ZHFT13]. A unified and comprehensive description is given in Chapter 5 (cf. Figure 1.1). Selected scheduling strategies have been evaluated to show that the proposed hierarchical model can be used to represent a wide range of scheduling strategies in principle. In particular, the periodic partial order scheduling strategy evaluated in Chapter 5 has been presented in [ZHF+13]. The quasi-static scheduling strategy evaluated in Chapter 5 is mainly driven by Falk [FKH+08; FZK+11; FZHT11; FZHT13]. Thus, only a brief summary is given.

Model-Based Optimizations at Lower Levels of Abstraction

In the proposed seamless model-based design flow, the hierarchical dataflow model at system level constitutes the input model to subsequent hardware/software syn-

thesis steps at the next lower levels of abstraction. Thus, a wide range of scheduling strategies can be synthesized in principle. More importantly, complex model-based optimizations can still be applied during synthesis at these lower levels of abstraction. As these optimizations can be automatically performed based on the proposed dataflow model, the overall modeling complexity is greatly reduced. Moreover, the proposed model-based optimizations considerably extend the design space, as different configurations can be automatically synthesized and evaluated during design space exploration in principle. The hardware/software synthesis of the proposed dataflow model, as well as model-based optimizations are discussed in Chapter 6 (cf. Figure 1.1). The overall synthesis framework has been presented in [ZHFT12b]. The micro-architectural optimizations described in Chapter 6 have also been presented in [ZHFT12b]. The inter-process resource sharing approach has been introduced in [ZHFT12a] for a small number of actor instances, and has been extended in [ZHF+14] to accommodate a large number of actor instances. The required modeling extensions have been presented in [ZFH+10].

2 Motivation

This chapter motivates the proposed dataflow model. To this end, Section 2.1 first reviews the typical steps in the design of digital systems from the specification to an implementation. In this context, the levels of abstraction from system level to gate level are described. For example, at system level, it is decided which parts of the application are to be computed in software, and which parts are to be computed in hardware. Here, multiprocessor System-on-Chip (MPSoC) architectures are becoming more and more important as implementation platforms for embedded systems due to the ever-increasing number and computational demands of functions performed by embedded software. These heterogeneous platforms are typically composed of *computing resources* like microprocessors, dedicated hardware accelerators, or analog and mixed-signal components, *storage resources* like random-access memory (RAM) or flash memory, as well as *communication resources* like buses, crossbars, and networks on chip (NoCs). However, without a design flow supporting these emerging complex platforms, the designer has to perform the *system synthesis* in an ad-hoc manner [Mar06]. Basically, system synthesis consists of two steps: First, it is decided which parts of the application should be bound to which resource available in the platform. However, this may result in resource contention, which must be resolved by a *temporal scheduling* of parts bound to the same resource. Second, the binding and scheduling decisions must be incorporated into the application.

Thus, at system level, a programming model which supports this synthesis process is desirable. Here, dataflow models of computation (MoCs) can be a solution. In a dataflow model, concurrent modules communicate via packets transmitted over channels. The dataflow model proposed in this thesis is described in Chapter 3. However, while this approach cleanly separates computation from communication, it is still unclear how the behavior of modules should be described. For the purpose of system synthesis, modules are typically constrained to a *functional* behavior, i.e., they should be platform-independent. Then, this *separation of concerns* (i.e., the separation of computation from communication and the separation of functionality from architecture) allows different bindings of the modules to the resources available in the platform to be automatically evaluated during design space exploration [KMN+00; KSS+09].

After a binding of modules to resources has been determined, modules bound to the same resource must be scheduled, as they cannot be executed concurrently. Here, a possible solution consists in generating a multi-threaded implementation, where each module is executed by a dedicated thread. This approach is outlined in Section 2.2. However, in order to reduce the scheduling overhead, a single-threaded implementation may be desirable. To this end, a *formal model* which provides for the *partial evaluation* of a module can be extracted from the functional behavior of each module. Then, multiple modules can be executed by a single thread in principle. This approach is outlined in Section 2.3.

After all binding and scheduling decisions have been made, the first step in system synthesis is finished. Subsequently, the made decisions must be incorporated into the dataflow model. For the proposed dataflow model, this second step in system synthesis is described in more detail in Chapter 5.

2.1 System Design and Levels of Abstraction

Figure 2.1 gives an abstract view on the design process of digital systems [KDWV02]: The starting point corresponds to the specification which has the maximal number of degrees of freedom. In order to transform the specification into an implementation, these degrees of freedom must be eliminated by *decision-*

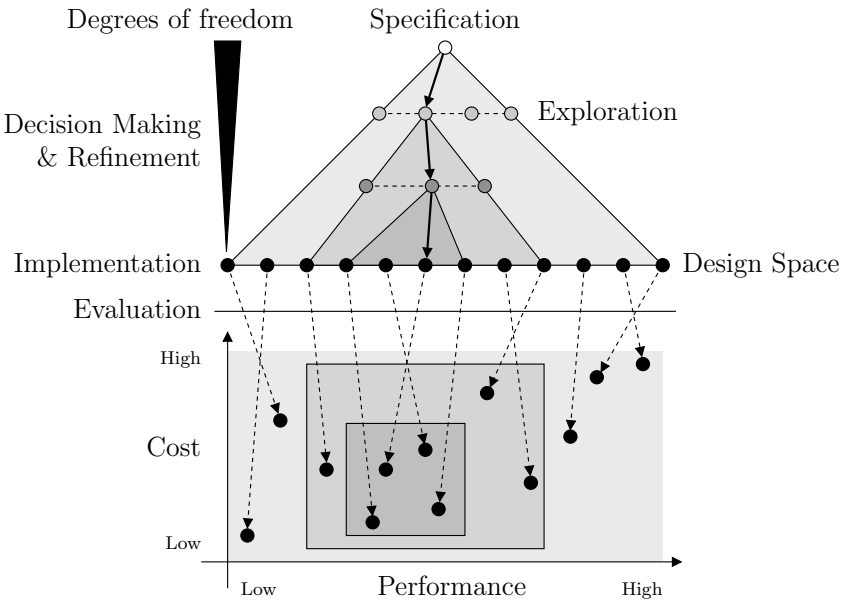


Figure 2.1: Starting from the specification, the degrees of freedom are eliminated by subsequent decision-making and refinement steps until an implementation is reached [KDWV02].

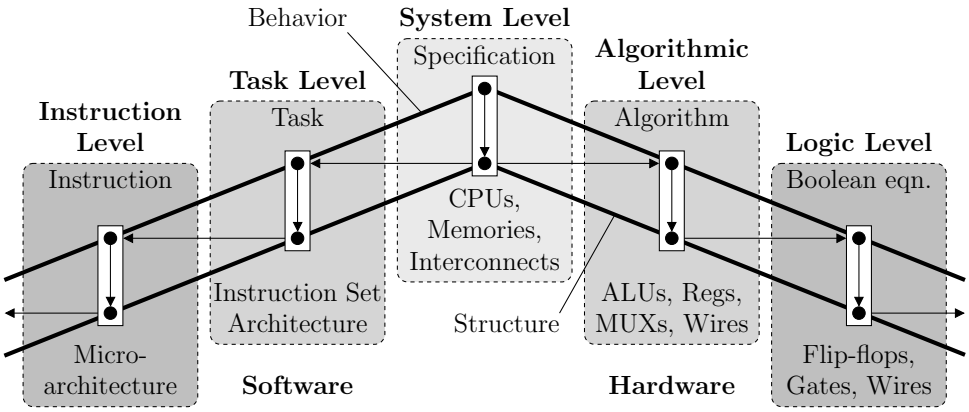


Figure 2.2: The double roof model defines a concrete top-down design process for digital hardware/software systems [TH07; GHP+09].

making and *refinement* [GHP+09]. At any level of abstraction, the input to the decision-making process is a *behavioral* model. The typical problems which must be solved by decision-making are (i) the allocation of resources, (ii) the binding of elements comprising the behavioral model to allocated resources, and (iii) the scheduling of elements bound to the same resource [TH07]. Subsequently, the made decisions are incorporated into the behavioral input model by a *refinement* step in order to obtain a *structural* output model. Together, decision-making and refinement are known as *synthesis* which must be performed at each level of abstraction. Note that a component of the structural output model corresponds to the behavioral input model to the synthesis step at the next lower level of abstraction.

The *double roof model* [TH07; GHP+09] defines a concrete top-down design process for digital hardware/software systems. The left-hand side of the model corresponds to the software design process, while the right-hand side corresponds to the hardware design process. For each side, different levels of abstraction can be identified. In turn, for each level of abstraction, the vertical arrow represents the synthesis step (i.e., decision-making and refinement) which must be performed in order to transform a behavioral input model into a structural output model. In contrast, the horizontal arrows represent the selection of a component of the structural output model to be used as the behavioral input model to the synthesis step at the next lower level of abstraction. Note that a possible *layout roof* representing the physical geometry as proposed in [GK83; WT85] has been omitted in Figure 2.2 for the sake of clarity. While traditionally, layout information has been of minor importance at higher levels of abstraction, it is increasingly employed even at the system level [PDBB06; ZGDS07].

As outlined above, at *system level*, it is decided which parts of the application are to be computed in software, and which parts are to be computed in hardware.

For the latter, at the *algorithmic level*, the transition from an untimed algorithm to a cycle-accurate structural description is performed by *behavioral synthesis* (also known as *high-level synthesis* (HLS)). At *logic level*, the cycle-accurate register-transfer level (RTL) description of the algorithm is further translated by *logic synthesis* tools to obtain the netlist representing the algorithm at gate level. At the hardware side, another level of abstraction (not shown in Figure 2.2) is the *circuit level*, where the gates determined by logic synthesis are further translated into transistors.

For parts to be computed in software, a *compiler* is typically used at the *task level* to translate these parts from a high-level language like C/C++ or Java into the instruction set architecture (ISA) of the target processor. Subsequently, at the *instruction level*, the instruction set of programmable processors is realized in hardware. The resulting structural model of the microarchitecture is typically specified as an RTL description.

Note that this thesis is mainly concerned with the higher levels of abstraction, namely the system level, the algorithmic level, and the task level. Chapter 3 presents the basic dataflow model which is used as input model at system level. The proposed dataflow model is expressive enough to model dynamic dataflow (DDF) applications. As a consequence, design methods applicable to less expressive models, like the static scheduling of modules bound to the same resource, cannot be used in the presence of such a highly expressive model. In order to use this potential, modules which adhere to less expressive models must be identified. This is discussed in Chapter 4. Note that this analysis is required in order to support the decision-making process at system level. The decision-making process itself as part of design space exploration at system level is not discussed in the context of this thesis. Subsequently, Chapter 5 discusses the refinement step at system level, i.e., the incorporation of binding and scheduling decisions into the dataflow model. Finally, Chapter 6 describes the synthesis step at the algorithmic and task levels. In particular, a synthesis framework is presented which provides for the hardware/software synthesis of the proposed dataflow model.

Figure 2.1 illustrates the importance to start a design flow at higher levels of abstraction: The more degrees of freedom have already been eliminated by previous synthesis steps, the smaller the influence of the remaining synthesis steps becomes w.r.t. the *quality indicators* of the design like cost, performance, or power consumption. In turn, this means that the possible design alternatives at each level of abstraction are covering increasingly smaller parts of the overall design space, as illustrated in the lower part of Figure 2.1. For example, the four possible design alternatives depicted in the center may be the result from providing different optimization flags to a compiler, resulting only in a minor trade off between throughput and code size. Obviously, at this level of abstraction, it is no longer possible to significantly improve the overall performance, which

would instead require to go back to higher levels of abstraction, and, for example, bind some parts of the application to hardware instead of software. However, the later in the design process such decisions are made, the more time-consuming (and thus expensive) the required changes become.

Thus, it is not only important to start a design flow at higher levels of abstraction in order to not exclude parts of the design space in the first place, but also to obtain early estimates of the quality indicators by evaluating the design alternatives preferably without having to construct the final product. In principle, this allows design alternatives to be selected during the decision-making process such that back-tracking to higher levels of abstraction is avoided.

For lower levels of abstraction, the synthesis steps are mostly well established by existing methodologies and tools. For example, at the software side, compilers for high-level languages have been researched for decades, and usually generate effective and efficient machine code. Analogously, on the hardware side at logic level, logic synthesis tools are in a similar position. At the algorithmic level, HLS tools are used by more and more industrial design flows, as can be seen by the broad availability of commercial HLS tools [MVG+13; For14; Cad14; Cal14; NEC14; Xil14]. These state-of-the-art HLS tools are usually able to transform a module into an equivalent RTL description if certain design rules are met. For example, dynamic memory management or recursion is typically not supported.

At system level, *system description languages* are still an active topic of research. Such a language should (i) provide for the functional description of the application in order to enable early functional validation, (ii) support automatic decision-making by providing for early quality estimation as outlined above, and (iii) provide for automatic refinement to a lower level of abstraction, as manually refining the application is usually time-consuming and error-prone.

Existing high-level languages like C/C++ are often used for task (i). However, due to the sequential nature of these languages, concurrency cannot be expressed adequately, rendering such languages unsuitable for task (ii) if the application is mapped to a platform with multiple computing resources, which is often the case in modern MPSoCs.

In order to address the shortcomings of sequential programming languages, various approaches have been proposed. For example, MATLAB/Simulink can be used to model, simulate, and analyze multidomain dynamic systems [BC12]. However, the inherent notion of simulation time makes MATLAB/Simulink models less suitable for task (i). In contrast, *system description languages* like SpecC [GZD+00] and SystemC [GLMS02] provide for the untimed and timed simulation of models. In all approaches, a high-level model (HLM) basically consists of a set of concurrent *modules* which communicate with each other via *channels*.

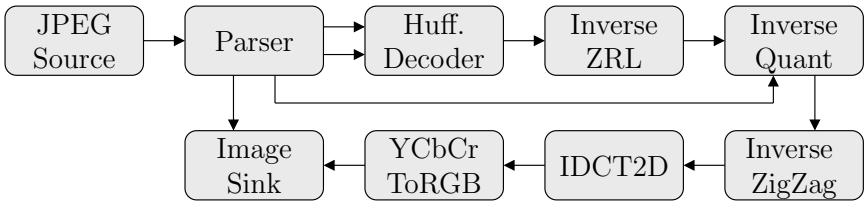


Figure 2.3: HLM of a JPEG decoder

Example 2.1. An example HLM is depicted in Figure 2.3, which shows the modules and channels comprising a JPEG decoder. While the shaded vertices represent the modules, the edges connecting the modules represent channels with first in, first out (FIFO) semantics. In the following, the data transmitted via FIFO channels are referred to as *tokens*.

In case of SpecC and SystemC, each module encapsulates one ore more *processes* as known from hardware description languages like VHDL or Verilog. In this case, the concurrent execution of processes is typically simulated by means of a discrete-event (DE) simulation. Note that instructions comprising a single process are always executed sequentially.

Example 2.2. Figure 2.4 shows the implementation of the `InverseQuant` module from Figure 2.3. Note that the module consists of two input ports (`infoIn` and `dataIn`), one output port (`dataOut`), and a single process (`thrd`) declared by the `SC_THREAD` macro. The `InverseQuant` module performs the inverse quantization of the DC/AC coefficients comprising a minimum coded unit (MCU) (also known as *macroblock*) by multiplying each coefficient with the corresponding entry of the quantization table `quantTable`. To this end, the `InverseQuant` module first reads the number of MCUs comprising the image, followed by the quantization table for the image.

As implemented, the `InverseQuant` module assumes that an MCU consists of $8 \times 8 = 64$ coefficients, which in turn consist of exactly one DC coefficient corresponding to the average brightness of the MCU, and 63 AC coefficients corresponding to the brightness variations across the MCU. Note that the DC/AC coefficients are computed by a discrete cosine transform (DCT) from the brightness values of the original MCU. The `IDCT2D` module reverses this operation.

Initially, an HLM based on a system description language like SpecC or SystemC is typically constrained to a *functional* behavior, i.e., it should be platform-independent. In order to obtain a timed model suitable for early quality

```
1  SC_MODULE(InverseQuant) {
2  public:
3    sc_fifo_in<int> infoIn;
4    sc_fifo_in<int> dataIn;
5    sc_fifo_out<int> dataOut;
6
7    SC_CTOR(InverseQuant) {
8      SC_THREAD(thrd);
9    }
10
11 private:
12   int quantTable[64];
13
14   void thrd() {
15     while(true) {
16       // Read the number of MCUs for the next frame
17       int mcuCount = infoIn.read();
18       // Read the quantization table
19       for(int i = 0; i < 64; ++i) {
20         quantTable[i] = infoIn.read();
21       }
22       // Process the MCUs
23       for(int m = 0; m < mcuCount; ++m) {
24         for(int j = 0; j < 64; ++j) {
25           dataOut.write(dataIn.read() * quantTable[j]);
26         }
27       }
28     }
29   }
30 };
```

Figure 2.4: Implementation of the `InverseQuant` module from Figure 2.3

estimation, the HLM is typically augmented with architecture information, i.e., number and type of processing and communication resources including buses and memories. By specifying the HLM binding to this architecture, and by using estimates for execution times, so called *Virtual Architecture Models* can be generated, permitting a combined functional and timed simulation, and hence, an early quality estimation [BBDV06; SFH+06; HHP+07; SGHT09; OGM+11; GCKR12]. In turn, this *separation of concerns* (i.e., the separation of computation from communication and the separation of functionality from architecture) directly supports the automatic decision-making at system level, as different

bindings of the HLM to the architecture can be easily evaluated during design space exploration [KMN+00; KSS+09].

However, the scheduling of modules bound to the same resource, which is also part of the decision-making process as outlined above, is more difficult to achieve for HLMs based on system description languages like SystemC. Here, a possible solution consists in generating a multi-threaded implementation, where each module is executed by a dedicated thread. This approach is outlined in Section 2.2. However, in order to reduce the scheduling overhead, a single-threaded implementation may be desirable. To this end, a formal model is typically extracted from the functional behavior of each module which provides for the partial evaluation of each module. Then, multiple modules can be executed by a single thread in principle. This approach is motivated in Section 2.3. Moreover, such a formal model enables the application of model-based analysis and optimization techniques as described in Chapter 4 and Chapter 6. In principle, both scheduling approaches could be evaluated during design space exploration in order to support the automatic decision-making at system level.

Task (iii), namely the refinement of the model by incorporating binding and scheduling decisions, is discussed in the context of this thesis solely for the formal dataflow model proposed in Chapter 3. Basically, binding and scheduling decisions are represented within the model by means of hierarchical modules. This refinement approach is discussed in more detail in Chapter 5.

2.2 Multi-Threaded Scheduling

Once a binding of modules to resources has been determined, the modules bound to the same resource must be scheduled. As each module consists of a set of processes, a possible solution would be to transform the model into a multi-threaded software implementation. As SystemC also provides synchronization channels like mutexes and semaphores, communication between processes via shared memory could be modeled and synthesized (although implementations based on these synchronization mechanisms are usually error-prone and thus not advisable in the general case [Lee06]). Then, for channels, a thread-safe implementation can be provided, and multiple threads can be executed on the same processor by means of a *preemptive multitasking* scheme. Here, each thread is guaranteed a slice of processor time in regular intervals. However, this approach requires an operating system which supports such a multitasking scheme. While readily employed in desktop PCs, the use of such an operating system may be infeasible in embedded systems due to limited resources.

A possible solution is to perform *cooperative multitasking*, i.e., tasks are executed until they relinquish control to the governing process. Incidentally, this is also the approach used by SystemC. Here, the governing process is the simulation

kernel, while a task corresponds to a process of a module. Each time a process calls `wait(e)`, control is handed back to the simulation kernel, and the process is suspended until the specified *event* `e` is signaled. Note that in Figure 2.4, the `wait` calls are hidden in the `read` and `write` methods of the input/output ports: If the attached input FIFO channel is empty, the process is suspended until at least one token is available. Likewise, if the attached output FIFO channel is full, the process is suspended until at least one free place is available. Methods of channels which may call `wait` are known as *blocking methods*, because they may suspend the process.

The key observation here is that it is safe to suspend a process when a `wait` statement (or a blocking channel method) is encountered. Then, once a process has been suspended, a different process may be executed on the same processor. Basically, a process then corresponds to a *coroutine*, which generalize subroutines by allowing multiple entry points for suspending and resuming execution at certain locations. In particular, each coroutine has associated its own stack pointer, so that stack variables are still available when a coroutine is resumed. In principle, coroutines can be realized by means of *user threads* (also known as *fibers*) [Sch13].

Coroutines must be scheduled by the user. In the case of SystemC, the DE simulation kernel uses an event list which is sorted by the simulation time when the events will be signaled. Each event has a set of associated processes which should be activated when the event is signaled. For software synthesis, simulation time could be ignored, thereby eliminating the need of a sorted event list. However, some scheduling overhead caused by event management and context switching remains which cannot be eliminated.

Example 2.3. Table 2.1 compares the different multi-threading strategies based on the `InverseQuant` module from Figure 2.3. In order to obtain a complete model, source and sink modules have been connected to the `InverseQuant` module by means of three FIFO channels. All three modules have been bound to the same core of an Intel Xeon E7-8837 processor with a clock frequency of 2.67GHz. Linux is used as operating system. It can be observed that the preemptive multitasking implementation based on POSIX threads (Pthreads) suffers from the fine granularity of the model, i.e., the amount of communication far outweighs the amount of computation performed by the `InverseQuant` module. Thus, the performance is significantly degraded by the thread-safe FIFO channel implementation, in addition to the expensive context switches incurred by the preemptive multitasking scheme. In contrast, the SystemC simulation performs much better than the Pthreads implementations: First, thread-safe channels are not required. Second, the scheduling overhead is reduced by the cooperative multitasking scheme used by SystemC, which is implemented

FIFO channel size	Multi-Threaded			Single-Threaded		
	Pthreads	SystemC	QT	DS	QSS	Merged
1	138440	10150	1861	384	124	54
2	135390	5730	1114	381	-	-
4	135930	3680	850	356	-	-
8	135190	2560	520	342	-	-
16	126676	2000	418	333	-	-
32	126286	1740	378	352	-	-
64	124744	1570	331	340	-	-
128	126742	1550	336	360	-	-
256	126478	1620	318	333	-	-
512	123250	1460	312	332	-	-
1024	125644	1550	309	333	-	-

Table 2.1: Latency (in ms) for different implementations of the `InverseQuant` module from Figure 2.3 with additional source and sink modules. The number of MCUs per frame has been set to 400000, and the given numbers correspond to the average latency of 10 frames.

by means of the QuickThreads framework [Kep93]. Note that for larger sizes of the FIFO channels, less context switches have to be performed, thereby reducing scheduling overhead and improving the performance. By eliminating the scheduling overhead imposed by the DE simulation kernel of SystemC, a minimal implementation also based on the QuickThreads framework further improves the latency by a factor of 5 for this example (cf. column "QT" in Table 2.1).

2.3 Single-Threaded Scheduling

In order to further reduce the scheduling overhead, the multi-threaded implementation may be transformed into a single-threaded implementation. Compared to the cooperative multitasking approach, this mainly requires the elimination of the stack pointer associated with each process. To this end, an analytical model which provides for a partial evaluation of processes must be extracted from the process-based model. For example, such a model could be based on a finite state machine (FSM), where the *states* (in the following called *modes*) represent conditions in which the process is waiting for missing resources (e.g., tokens or free places on FIFO channels), while the *transitions* represent instruction sequences which can be executed without blocking if enough resources are available. After such an analytical model has been extracted from the process-based model,

the individual stack pointers can be eliminated by transforming the *live stack variables* of each transition into member variables of the module. Note that live variables of a transition are variables that may be read by another (or the same) transition before they are written. Given an FSM as described above, the live variables can be determined by performing a classic *dataflow analysis* [Kil73].

In order to transform a process into an FSM as described above, an intuitive approach first identifies all blocking channel method calls. Then, for each blocking channel method call, a mode is allocated in which the process may wait for missing resources corresponding to the blocking channel method call. Consequently, a `return` statement is inserted into the process before each blocking channel method call. In this way, the process is exited before a blocking channel method is called, thereby allowing the scheduling code to execute transitions of other processes. When enough resources are eventually available such that the blocking channel method can be executed without blocking, we must be able to return to the point of exit, i.e., the blocking channel method call. For example, this can be achieved by associating a `case` label with the blocking channel method call, and wrapping the process in a `switch` statement. Note that the blocking channel method call corresponds to the first instruction associated with a transition.

Let $M = \{m_0, \ldots, m_n\}$ be the set of modes allocated by the proposed transformation. Then, it can be observed that for the FSM derived from any process by this transformation, by construction, a mode $m_i \in M$ has exactly one outgoing transition t_i corresponding to the sequence of instructions from the blocking channel method call associated with m_i to a `return` statement associated with another mode m_j. However, due to complex control flow of the process, different `return` statements may be reached from the blocking channel method call in principle. Thus, a transition t_i may have a set of possible target modes $M'(t_i) \subseteq M$ from which the concrete target mode is selected by the transition during its execution. For each transition t_i, the set of possible target modes $M'(t_i)$ can be efficiently determined, e.g., by means of a depth-first search (DFS) of the control flow graph (CFG) of the transformed process starting from the first instruction of t_i, i.e., the blocking channel method call.

Example 2.4. The result of this transformation for the `InverseQuant` module from Figure 2.4 is shown in Figure 2.5. As there are four blocking channel method calls (cf. line 12, line 16, line 22, and line 25), we first allocate four modes m_1–m_4 where the process is blocked waiting for resources. Note that an additional mode m_0 is allocated for one-time initialization code. Remember that each mode $m_i \in \{m_0, \ldots, m_4\}$ has exactly one outgoing transition $t_i \in \{t_0, \ldots, t_4\}$. For example, the sole outgoing transition t_3 of mode m_3 corresponds to lines 22–23, and the only target mode of t_3 is m_4, i.e., $M'(t_3) = \{m_4\}$. In contrast, knowing that the loops with constant lower and upper bounds

```
1  class InverseQuantFsm {
2  public:
3    ...
4    enum Mode { M0, M1, M2, M3, M4 };
5
6    template<Mode mode> Mode thrdFsm() {
7      switch(mode) {
8      case M0:
9        while(true) {
10         return M1;
11     case M1:
12         mcuCount = infoIn.read();
13         for(i = 0; i < 64; ++i) {
14           return M2;
15     case M2:
16           quantTable[i] = infoIn.read();
17         }
18         for(m = 0; m < mcuCount; ++m) {
19           for(j = 0; j < 64; ++j) {
20             return M3;
21     case M3:
22             tmp = dataIn.read() * quantTable[j];
23             return M4;
24     case M4:
25             dataOut.write(tmp);
26           }
27         }
28       }
29     }
30   }
31
32 private:
33   int quantTable[64];
34   int mcuCount, i, m, j, tmp;
35 };
```

Figure 2.5: Transformed `thrd` process from Figure 2.4.

(lines 13 and 19) are executed at least once, the set of possible target modes for transition t_2 of mode m_2 is $M'(t_2) = \{m_1, m_2, m_3\}$.

The stack variables `mcuCount`, `i`, `m`, and `j` (cf. Figure 2.4) have been identified as live variables and consequently have been transformed into member variables of the `InverseQuant` module. Note that an additional variable `tmp`

has been allocated which stores the multiplied DC/AC coefficient (cf. line 22) until it can be written on the output port `dataOut` (cf. line 25).

Note that the process has been transformed into a C++ *template method* which must be parameterized by the desired mode m_0–m_4. For example, by instantiating the template method by means of a function call `thrdFsm<M1>`, a compiler should be able to eliminate the `switch` statement, and only retain the instructions associated with the only outgoing transition of m_1, i.e., t_1.

The sequence of instructions associated with a transition is called *action function* in the following. An action function can be invoked (or *fired*) only if enough resources are available. To this end, the condition which determines whether the action function may be fired or not is annotated as a *guard predicate* to the transition. As each action function is associated with at most one blocking channel method call according to the proposed transformation, each transition also has at most one guard predicate pertaining to resources required to execute the action function. Note that Chapter 3 extends transitions to more general guard predicates which also permit the annotation of resource requirements for more than one channel, as well as the evaluation of member variables and tokens on input ports.

Example 2.5. The FSM resulting from analyzing the transformed `thrd` process from Figure 2.5 is shown in Figure 2.6. Note that *pseudostates* (represented by the small black circles) are used to visualize the transitions with more than one possible target mode (cf. Example 2.4). Transitions are shown in the form "t_i: guard predicate/action function". For example, the guard predicate of transition t_1 consists of "#`infoIn` ≥ 1", which encodes that at least one token is required on input port `infoIn` in order to execute the associated action function `thrdFsm<M1>`.

Having determined an FSM for each process of the example, the transitions of the FSMs must be scheduled. A possible approach is to dynamically schedule the transitions. Dynamic scheduling performs all scheduling decisions at run time. Here, one possibility is to evaluate the modules in a round-robin fashion, moving on to the next module if no more transitions of the current module can be executed.

Example 2.6. Results for dynamically scheduling the transitions of the FSMs are given in column "DS" in Table 2.1. It can be observed that for small sizes of the FIFO channels, the dynamically scheduled single-threaded implementation outperforms the multi-threaded implementation based on QuickThreads by

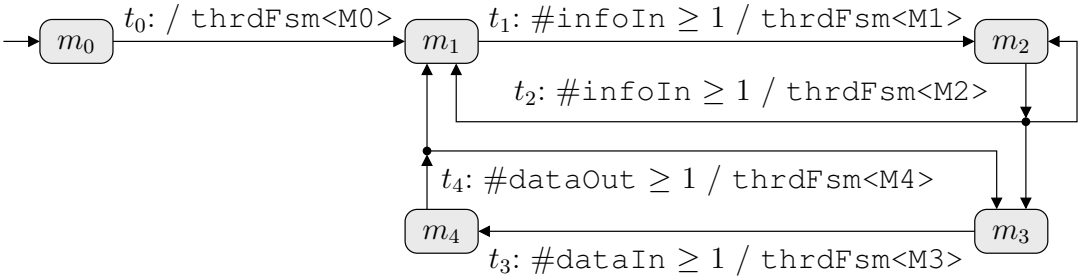


Figure 2.6: FSM representing the communication behavior of the transformed `thrd` process from Figure 2.5. Pseudostates (small black circles) are used to visualize transitions with more than one possible target mode.

a factor of approximately 5. This is due to the fact that the single-threaded implementation eliminates all context switches still required for the cooperative multitasking scheme implemented by means of the QuickThreads framework. For larger sizes of the FIFO channels, the dynamically scheduled single-threaded implementation performs slightly worse than the multi-threaded implementation based on QuickThreads due to the scheduling overhead incurred by the former.

In order to further reduce the overhead of the dynamic scheduling scheme, a quasi-static scheduling scheme has been evaluated. A quasi-static schedule (QSS) executes static transition sequences (that have been determined at compile time) based on data-dependent scheduling decisions that are made at run time. At a glance, the transitions of the modules are matched, similar to the scheme presented in [PSB09a]. This approach results in a QSS which evaluates the current modes of the generated FSMs at run time, but is then able to execute static transition sequences without having to make intermediate scheduling decisions. For example, a transition t_p which produces a token onto a channel should have a corresponding transition t_c which consumes the token from the channel. In this case, the static transition sequence $\langle t_p, t_c \rangle$ can be executed at run time without having to make intermediate scheduling decisions.

Example 2.7. Consider again Figure 2.6: In order to execute transition t_1, one token is required from input port `infoIn`. Then, one has to find a transition of the source module which produces the corresponding token, and so on. The longest static transition sequence found in this way consists in the production of a DC/AC value by the source module, followed by transitions t_3 and t_4 of the `InverseQuant` module which transform and forward the DC/AC value, and finally the consumption of the DC/AC value by the sink module.

Results for quasi-statically scheduling the transitions of the FSMs are given in column "QSS" in Table 2.1. Note that increasing the sizes of the FIFO

channels makes no sense in this case, because the maximal buffer size required by the calculated QSS is 1. Thus, increasing the sizes of the FIFO channels is, in fact, detrimental to the performance (not shown). It can be observed that the quasi-statically scheduled implementation outperforms the dynamically scheduled implementation by a factor of approximately 3.

A further reduction of scheduling overhead can be achieved by manually merging the processes bound to the same processor core into a single process. In particular, this allows the compiler to perform some more optimizations and improves cache locality.

Example 2.8. Results for the manually merged processes are given in the rightmost column "Merged" in Table 2.1. It can be observed that this approach outperforms the quasi-statically scheduled single-threaded implementation by a factor of approximately 2.3.

In general, the influence of the scheduling overhead on the overall performance is reduced the more coarse-grained the modules become, i.e., the more computation is performed in relation to communication. Thus, choosing the right granularity of modules is a challenging task. If the HLM consists only of a small number of coarse-grained modules, the parallelism offered by the platform may not be utilized to its full potential. Here, automatic parallelization at task level has been studied for decades but shows success only in limited areas like loop parallelization [GGL12]. Thus, it is a feasible approach to assume a large number of fine-grained modules, which then, however, induces the scheduling problem as outlined above.

To reiterate, the dataflow model based on transitions is formally introduced in Chapter 3, while the incorporation of scheduling decisions (like the QSS as outlined above) into the model is described in Chapter 5.

2.4 Related Work and Limitations

The transformation of a multi-threaded implementation to a single-threaded implementation has been studied before. The proposed approach is similar to the one presented in [FZK+11]. However, in [FZK+11], the resulting FSM is based on the CFG of the process. Thus, this results in an initially large FSM, as the complete control flow is reflected by the FSM. As a consequence, it follows that not all transitions perform a blocking channel method call. In order to alleviate this problem, [FZK+11] performs a post-processing step to merge some of the modes and transitions. In contrast, the proposed approach as outlined above generates a minimal number of modes (and transitions) w.r.t. the number of blocking

channel method calls (ignoring the single transition corresponding to one-time initialization code), thereby rendering a post-processing step superfluous.

Furthermore, the modes calculated by [FZK+11] may have (i) multiple outgoing transitions (with different communication behaviors), and (ii) transitions with more complex guard predicates which may evaluate member variables of the modules. By construction, each mode of the proposed approach only has one outgoing transition. Moreover, the guard predicate of each transition consists only of the number of resources required by the transition. Thus, scheduling overhead is reduced, as less transitions have to be evaluated in each mode, and no evaluation of member variables is necessary.

A similar approach is used by the *asynchronous programming* feature of C# 5.0 [BRM+12]. Here, *asynchronous* methods return *task* objects which implement a similar state machine as derived by the proposed approach. While this scheme allows a dynamic creation of tasks objects, the scheduling of transitions is tightly coupled with the resulting state machine, i.e., custom scheduling strategies are not supported. In contrast, the proposed approach supports custom scheduling strategies as outlined in the previous section. However, in the general case, dynamic creation of tasks is not supported by the proposed approach. Note that this also prohibits the use of recursive functions if blocking channel method calls are performed in such a function. However, as a module description should serve for both hardware and software synthesis, recursive functions usually must be avoided anyway, as HLS tools typically perform a similar transformation to obtain a single, static FSM.

The FSM generated by the proposed process transformation only reflects blocking channel methods calls. For SystemC FIFO channels, these are calls to the `read` and `write` methods. In contrast, non-blocking channel methods are not reflected, which may hinder analysis. However, it should be noted that these methods are implicitly supported by the proposed process transformation, as they can never block on missing resources. Concerning SystemC FIFO channels, the `nb_read` and `nb_write` methods implicitly check if at least one token or free place is available before consuming or producing the token. Thus, these methods can be trivially rewritten to utilize only the blocking `read` and `write` methods and the non-blocking `num_available` and `num_free` channel methods, which can be used to explicitly perform resource availability checks before consuming or producing tokens. While all token accesses would then be properly reflected by the FSM, the resulting transitions would still contain calls to the `num_available` and `num_free` channel methods. Moreover, similar resource availability checks are performed by the scheduling code, thereby effectively duplicating the resource availability checks. For the proposed dataflow model, however, eliminating the calls to the `num_available` and `num_free` channel methods is not feasible in the general case, mainly due to data-dependent control flow, and the fact that guard predicates cannot check the *absence* of resources.

3 Model

Due to the ever-increasing number and computational demands of functions performed by embedded software, multiprocessor System-on-Chip (MPSoC) architectures are becoming more and more important as implementation platforms for embedded systems. These heterogeneous platforms are typically composed of multiple microprocessors, dedicated hardware accelerators, analog and mixed-signal components, as well as interconnects like buses, crossbars, and networks on chip (NoCs). Thus, at system level, choosing the right programming model is a challenging task. Here, dataflow models of computation (MoCs) naturally expose the parallelism contained in the application, and thus yield well to synthesis for MPSoC platforms. In a dataflow model, concurrent modules communicate via packets transmitted over channels. The resulting model is also referred to as *dataflow graph* (DFG)in the following.

Figure 3.1 shows an exemplary design flow supported by the proposed dataflow model. Note that only the steps pertaining to system synthesis are shown. Initially, the desired behavior is specified by a DFG, where functionality is still separated from architecture. For example, the modules of the DFG could be derived from well-formed SystemC modules as described in Section 2.3, reused from a module library, or written from scratch according to the model semantics described in this

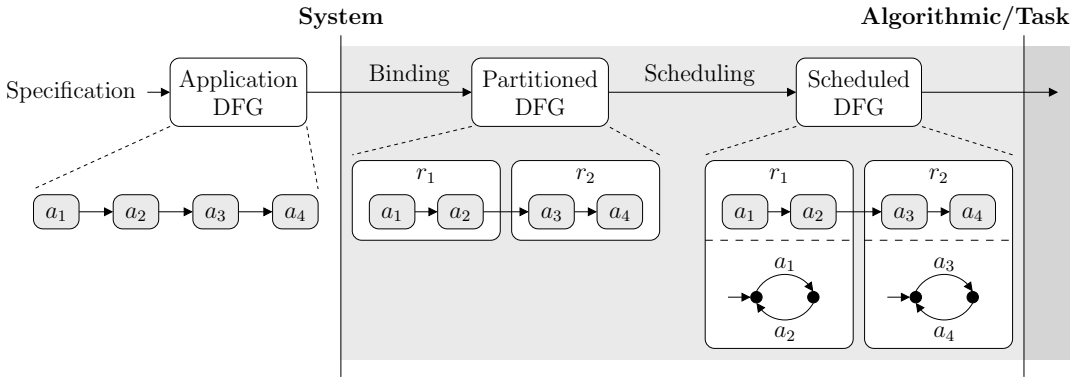


Figure 3.1: Design flow supported by the proposed model. Only the steps pertaining to system synthesis are shown (cf. Figure 2.2 on page 9).

chapter. This application DFG ("Application DFG") can be used, for example, to perform functional verification. Note that the application DFG corresponds to the non-hierarchical model which is discussed in this chapter.

After having determined a binding of modules to resources available in the MPSoC platform, the modules of the application DFG can be partitioned accordingly during the refinement step, resulting in a hierarchical DFG ("Partitioned DFG"). This step represents a structural refinement only, and modules are still executed in an unconstrained manner. Next, modules assigned to the same resource are scheduled ("Scheduled DFG"), as they cannot be executed concurrently. The incorporation of binding and scheduling decisions into the model is discussed in Chapter 5. The scheduled DFG serves as input to subsequent synthesis steps of the design flow at lower levels of abstraction, namely the task level and the algorithmic level. These synthesis steps are discussed in Chapter 6.

In Section 3.1, the basic (i.e., non-hierarchical) dataflow model and its operational semantics are described. In Section 3.2, the basic model is extended by hierarchical finite state machines (FSMs) as known from Statecharts [Har87] in order to cope with the complexity found in real-world applications. This extension has been used to model the running example from the networking domain in Chapter 6. Finally, Section 3.3 shows that the model, if restricted to guard predicates which specify resource requirements only, is as expressive as the unrestricted model, where guard predicates are also allowed to evaluate member variables and tokens on input ports. Note that the restricted dataflow model basically corresponds to the formal model which can be extracted from well-formed SystemC modules as described in Section 2.3. Whereas Section 3.1 is required for the following chapters, Section 3.2 and Section 3.3 can be skipped in principle.

3.1 Non-Hierarchical Model

In this section, the basic (i.e., non-hierarchical) dataflow model as introduced in [ZHF+13; ZHFT13] is described. In order to enable model-based transformations (including the scheduling of modules as outlined in Section 2.3), the behavior of a module can be described by a set of *firing rules* [LP95]. In the following, modules described by firing rules are referred to as *actors*. In turn, firing rules are often described by means of *guarded actions* [RA04; BSS10]. A guarded action consists of a *guard predicate* and an *action function*, and the guard predicate determines whether the action function can be executed or not. Guarded actions are particularly suited do describe dynamic dataflow (DDF) applications, where actor firings may depend on the availability of tokens on different channels, or on the values of actor variables. However, besides DDF actors, firing rules can also be used to model static dataflow actors. For example, if all firing rules

have the very same token consumption and production rates, the actor can be classified as a synchronous dataflow (SDF) [LM87] actor. Chapter 4 discusses the identification and representation of less expressive dataflow models in more detail.

In the following, non-hierarchical actors are called *leaf actors*, while hierarchical actors are called *composite actors*. The discussion of the hierarchical model is postponed to Section 5.1, as it is mainly used to reflect binding and scheduling decisions. It should be noted that the set of leaf actors is a strict subset of the set of composite actors, i.e., a leaf actor is also a valid composite actor.

At a glance, leaf actors consist of a structural part (ports), and a behavioral part (transitions). More formally, leaf actors are defined as follows:

Definition 3.1 (Leaf actor)**.** A leaf actor $a = (I, O, M, m_{\text{cur}}, m_0, v, v_0, F_g, F_a, T, \text{peek}, \text{cons}, \text{prod})$ consists of a set of input ports I, a set of output ports O, a set of actor modes M, a current actor mode $m_{\text{cur}} \in M$, an initial value $m_0 \in M$ for m_{cur}, an n-tuple of actor variables $v = (v_1, \ldots, v_n) \in \mathbb{Z}^n$ representing some internal actor state, initial values $v_0 \in \mathbb{Z}^{|v|}$ for v, a set of guard functions F_g representing general predicates on token values or actor variables, a set of action functions F_a, a set of transitions T implementing the behavior of the actor, a function $\text{peek}\colon I \times F_g \to \mathbb{N}_0$ which specifies for each input port $p \in I$ and guard function $f_g \in F_g$ the number of tokens evaluated by f_g, a function $\text{cons}\colon I \times F_a \to \mathbb{N}_0$ which specifies for each input port $p \in I$ and action function $f_a \in F_a$ the number of tokens consumed by f_a, and a function $\text{prod}\colon O \times F_a \to \mathbb{N}_0$ which specifies for each output port $p \in O$ and action function $f_a \in F_a$ the number of tokens produced by f_a.

In principle, actor ports are connected by point-to-point channels with first in, first out (FIFO) semantics. As the channels are part of the hierarchical actor model, the formal definition of the resulting topology is postponed to Section 5.1.

In the following, we abstract from complex token and variable types, and assume that they are integers, i.e., whole numbers. Note that m_{cur} can be considered as an additional actor variable in addition to v, which leads to the following definition of the overall *actor state*:

Definition 3.2 (Actor State)**.** The *actor state* $s = (v, m_{\text{cur}})$ consists of the actor variables v and the current actor mode m_{cur}. Consequently, the *initial actor state* corresponds to $s_0 = (v_0, m_0)$.

Guard functions are associated with transitions, and determine whether a transition is enabled or not. To this end, they may evaluate the actor state and tokens on input ports. More formally, each guard function $f_g \in F_g$ is defined as follows:

Definition 3.3 (Guard function)**.** A guard function f_g is a function

$$f_g \colon \mathbb{Z}^{\text{peek}(i_1, f_g)} \times \cdots \times \mathbb{Z}^{\text{peek}(i_{|I|}, f_g)} \times \mathbb{Z}^{|v|} \times M \to \{\top, \bot\}$$

which evaluates sequences of input tokens and the actor state in order to determine whether the governing transition is enabled or not.

It should be noted that this definition of guard functions implies that a guard function cannot transform the actor state, nor consume or produce tokens. This means that they are side-effect free w.r.t. the state of the model. In contrast, *action functions* may transform the actor state, and usually consume or produce tokens. More formally, each action function $f_a \in F_a$ is defined as follows:

Definition 3.4 (Action function)**.** An action function f_a is a function

$$\begin{aligned} f_a \colon \mathbb{Z}^{\text{cons}(i_1, f_a)} \times \cdots \times \mathbb{Z}^{\text{cons}(i_{|I|}, f_a)} \times \mathbb{Z}^{|v|} \times M \to \\ \mathbb{Z}^{\text{prod}(o_1, f_a)} \times \cdots \times \mathbb{Z}^{\text{prod}(o_{|O|}, f_a)} \times \mathbb{Z}^{|v|} \times M \end{aligned}$$

which transforms sequences of input tokens into sequences of output tokens, and may additionally transform the actor state.

Transitions combine actor modes, guard functions, and action functions in order to form the *actor FSM*. More formally, transitions of leaf actors are defined as follows:

Definition 3.5 (Transition)**.** A transition $t = (m, M', f_g, f_a) \in T$ specifies the source mode $m \in M$ in which the transition is active (i.e., considered for evaluation), a set of possible target modes $M' \subseteq M$, a guard function $f_g \in F_g$, and an action function $f_a \in F_a$. A transition is *well-formed* in the proposed dataflow model if $\forall p \in I : \text{peek}(p, f_g) \leq \text{cons}(p, f_a)$. In the following, only well-formed transitions are considered.

Not all combinations of guard functions and action functions result in well-formed transitions in the proposed dataflow model: For a given transition, the guard function f_g is not allowed to peek at more tokens from any input port $p \in I$ than the associated action function f_a consumes, i.e., $\text{peek}(p, f_g) \leq \text{cons}(p, f_a)$. Thus, the values of cons correspond to the overall communication behavior of a transition, even if the guard function accesses tokens. This restriction allows us to abstract from the values of peek for analytic purposes, because they are subsumed by the values of cons.

It should be obvious from the above definitions that transitions in the proposed dataflow model exhibit a static communication behavior, i.e., when executed, a transition always consumes and produces the same number of tokens w.r.t. an actor port, and this number is statically known at compile time. While the

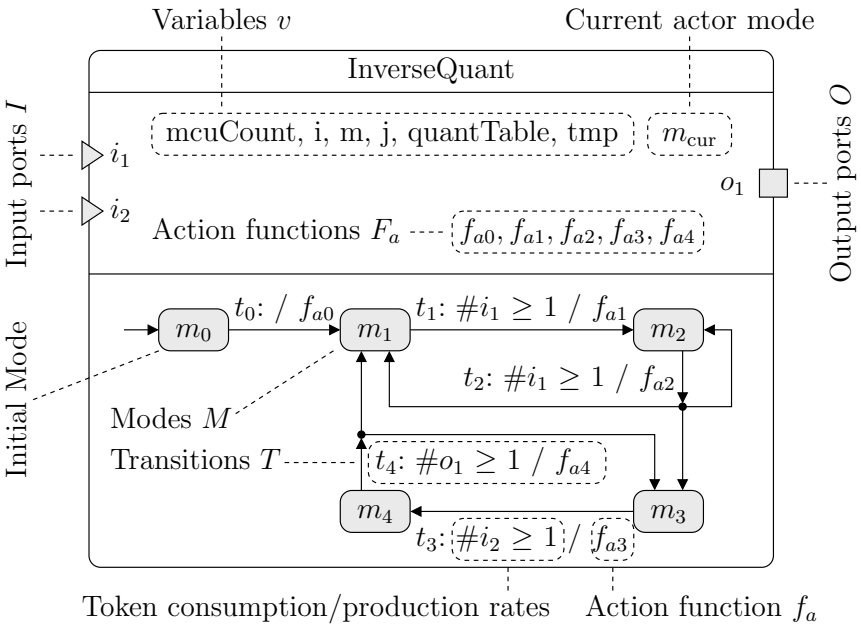


Figure 3.2: Components of the `InverseQuant` actor. Note that $F_g = \emptyset$ as per construction (cf. Section 2.3).

proposed dataflow model provides for DDF actors, this property ensures that less expressive dataflow MoCs can still be identified as described in Chapter 4.

In the following, the special guard function $f_\top$ is used to denote a guard function which does not peek at tokens from input ports (i.e., $\forall p \in I : \text{peek}(p, f_\top) = 0$), whose result does not depend on the actor state, and which always evaluates to $\top$: $f_\top(\langle\rangle, \ldots, \langle\rangle, v, m_{\text{cur}}) \mapsto \top$. Note that $\langle\rangle$ denotes the empty token sequence. The guard function $f_\top$ is implicitly added to the set of guard functions F_g of an actor, and permits to model transitions with no guard function.

Example 3.1. Figure 3.2 shows the `InverseQuant` actor from Figure 2.5 on page 18 according to the proposed dataflow model. It consists of input ports $I = \{i_1, i_2\}$, output ports $O = \{o_1\}$, variables $v = (\text{mcuCount}, \text{i}, \text{m}, \text{j}, \text{quantTable}, \text{tmp})$, and action functions $F_a = \{f_{a0}, f_{a1}, f_{a2}, f_{a3}, f_{a4}\}$. Note that, as per construction, $F_g = \emptyset$ (cf. Section 2.3). The initial values of the variables can be assumed to be $v_0 = \mathbf{0}$, although being derived from a SystemC thread, transition t_0 is typically used to reset the variables.

The shaded vertices in the lower part of Figure 3.2 correspond to the modes $M = \{m_0, \ldots, m_4\}$, with m_0 being the initial mode (cf. Figure 2.6 on page 20). Each mode m_i has one outgoing transition t_i, represented by the edges connecting the modes. Remember that transitions with more than one possible target

mode (t_2 and t_4 in this case) are visualized by pseudostates (small black circles). As per construction, no transition has an associated guard function, i.e., $\forall t \in T : t.f_g = f_\top$. For the sake of clarity, the annotation of $f_\top$ to the transitions in Figure 3.2 is omitted.

Concerning the token consumption and production rates, "$\#p \geq n$" annotated to a transition t with action function f_a specifies either $\text{cons}(p, f_a) = n$ if $p \in I$, or $\text{prod}(p, f_a) = n$ if $p \in O$, respectively. For the sake of clarity, the annotation of token consumption and production rates is omitted for a port p if $\text{cons}(p, f_a) = 0$ or $\text{prod}(p, f_a) = 0$, respectively. For example, the token consumption and production rates of f_{a1} are $\text{cons}(i_1, f_{a1}) = 1$, $\text{cons}(i_2, f_{a1}) = 0$, and $\text{prod}(o_1, f_{a1}) = 0$, which can be deduced by inspecting transition t_1 in Figure 3.2.

3.1.1 Operational Semantics

In general, actors transform sequences of input tokens into sequences of output tokens. To this end, $s(p)$ refers to the sequence of tokens currently available on a port $p \in I \cup O$, while $|s(p)|$ refers to the length of $s(p)$, i.e., the number of tokens currently available on a port $p \in I \cup O$.

As a transition t consists of a guard function $f_g \in F_g$ and an action function $f_a \in F_a$, the operational semantics of transitions can be split into two phases, namely an *evaluation phase* where f_g is evaluated, possibly followed by an *execution phase* where f_a is executed. The two phases are described in more detail in the following sections.

Evaluation phase

During the evaluation phase, a transition t is evaluated in order to determine whether it is enabled or not. The evaluation phase is summarized by Algorithm 3.1. Basically, t is enabled if the current actor mode matches $t.m$, enough resources are available in order to execute $t.f_a$, and the guard function $t.f_g$ evaluates to $\top$. In more detail, t is enabled if all of the following conditions are met:

- The current actor mode must be equal to the actor mode in which the transition is active, i.e., $m_{\text{cur}} = t.m$ (cf. lines 2–4). This condition prevents the execution of transitions which are not active according to the current actor mode.

- On each input port $p \in I$, at least $\text{cons}(p, t.f_a)$ tokens must be available, i.e., $\forall p \in I : |s(p)| \geq \text{cons}(p, t.f_a)$ (cf. lines 5–7). This condition prevents action functions from processing possibly invalid token values.

Algorithm 3.1 Evaluation phase of transitions

```
1: procedure EVALUATE(Transition t)
2:     if m_cur ≠ t.m then
3:         return ⊥
4:     end if
5:     if ∃p ∈ I : |s(p)| < cons(p, t.f_a) then
6:         return ⊥
7:     end if
8:     if ∃p ∈ O : |s(p)| + prod(p, t.f_a) > K(p) then
9:         return ⊥
10:    end if
11:    return t.f_g(s_head(i_1, t.f_g), ..., s_head(i_|I|, t.f_g), v, m_cur)
12: end procedure
```

- On each output port $p \in O$, at least $\text{prod}(p, t.f_a)$ free places must be available (cf. lines 8–10). Note that in contrast to SDF graphs, channels in our model are bounded, which is why we also perform space availability checks.[1] To reiterate, channels are part of the hierarchical actor model which is discussed in Section 5.1. Suffice to say that in the following, the function $\text{K}\colon O \to \mathbb{N}$ is used to denote the capacity of the FIFO channel bound to an output port $p \in O$. Then, enough free places are available on p if $|s(p)| + \text{prod}(p, t.f_a) \leq \text{K}(p)$. This condition prevents action functions from overwriting tokens which are not yet consumed from the channel.

- Finally, the guard function $t.f_g$ applied to the actor state and the token subsequences from the input ports according to the values of peek must evaluate to $\top$ (cf. line 11). To this end, on each input port $p \in I$, at least $\text{peek}(p, t.f_g)$ tokens must be available, i.e., $\forall p \in I : |s(p)| \geq \text{peek}(p, t.f_g)$. Note that for any input port $p \in I$, the availability of at least $\text{cons}(p, t.f_a)$ tokens also implies the availability of at least $\text{peek}(p, t.f_g)$ tokens, because according to Definition 3.5, $\forall p \in I : \text{cons}(p, t.f_a) \geq \text{peek}(p, t.f_g)$. Thus, (the sequential) Algorithm 3.1 does not explicitly perform these resource availability checks. If enough tokens are available on each input port $p \in I$, the sequence of available tokens $s(p)$ can be split into two subsequences $s_{\text{head}}(p, f_g)$ and $s_{\text{tail}}(p, f_g)$ such that[2] $s(p) = s_{\text{head}}(p, f_g)^\frown s_{\text{tail}}(p, f_g)$, and $|s_{\text{head}}(p, f_g)| =$

[1] Note that a bounded channel can be modeled by two unbounded channels where one of the channels contains a number of initial tokens equal to the capacity of the bounded channel. Thus, a model with bounded channels is not more expressive than a model without bounded channels.

[2] The sequence concatenation operator "$^\frown$" is defined for two sequences $a = \langle a_1, \ldots, a_n \rangle$ and $b = \langle b_1, \ldots, b_m \rangle$ as follows: $a^\frown b = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$.

peek(p, f_g). Then, the result of executing f_g is $r = f_g(s_{\text{head}}(i_1, f_g), \ldots, s_{\text{head}}(i_{|I|}, f_g), v, m_{\text{cur}})$ (cf. Definition 3.3). Thus, f_g evaluates to $\top$ if $r = \top$.

When all conditions are met, t is enabled and becomes eligible for *execution*, which is detailed in the next section.

Example 3.2. Consider transition t_1 from Figure 3.2. As $\text{cons}(i_1, f_{a1}) = 1$, $\text{cons}(i_2, f_{a1}) = 0$, and $\text{prod}(o_1, f_{a1}) = 0$ (cf. Example 3.1), only input port i_1 may not have enough tokens. Therefore, t_1 is enabled if $m_{\text{cur}} = t.m = m_1$ and $|s(i_1)| \geq \text{cons}(i_1, f_{a1}) = 1$. Note that the guard function $t.f_g = f_\top$ always evaluates to $\top$ according to its definition.

Execution phase

During the execution phase, the action function f_a of a transition t is invoked, thereby consuming and producing tokens and possibly modifying the actor state. The execution phase is summarized by Algorithm 3.2.

First, the action function f_a is applied to the actor state and the token subsequences from the input ports according to the values of cons (cf. line 2).

Remember that the evaluation phase ensures that enough tokens are available according to cons. Thus, analogously to the evaluation of guard functions, the token sequence $s(p)$ available at each input port $p \in I$ can again be split into two subsequences $s_{\text{head}}(p, f_a)$ and $s_{\text{tail}}(p, f_a)$ such that $s(p) = s_{\text{head}}(p, f_a)^\frown s_{\text{tail}}(p, f_a)$, and $|s_{\text{head}}(p, f_a)| = \text{cons}(p, f_a)$.

Furthermore, the evaluation phase ensures that enough free places are available according to prod. In the following, $s_{\text{prod}}(p, f_a)$, denotes the sequence of tokens produced by f_a on an output port $p \in O$, i.e., $|s_{\text{prod}}(p, f_a)| = \text{prod}(p, f_a)$.

Algorithm 3.2 Execution phase of transitions

```
1: procedure EXECUTE(Transition t)
2:     Let (s_prod(o_1, t.f_a), ..., s_prod(o_|O|, t.f_a), v', m') ←
           t.f_a(s_head(i_1, t.f_a), ..., s_head(i_|I|, t.f_a), v, m_cur)
3:     for all p ∈ I do
4:         s(p) ← s_tail(p, t.f_a)
5:     end for
6:     for all p ∈ O do
7:         s(p) ← s(p)^s_prod(p, t.f_a)
8:     end for
9:     v ← v'
10:    m_cur ← m'
11: end procedure
```

Then, the result of executing f_a is $(s_{\text{prod}}(o_1, f_a), \ldots, s_{\text{prod}}(o_{|O|}, f_a), v', m') = f_a(s_{\text{head}}(i_1, f_a), \ldots, s_{\text{head}}(i_{|I|}, f_a), v, m_{\text{cur}})$ (cf. Definition 3.4). Executing f_a has the following effects on the overall model state:

- For each input port $p \in I$, the sequence of currently available tokens $s(p)$ is shortened by the sequence of consumed tokens $s_{\text{head}}(p, f_a)$, i.e., $\forall p \in I : s(p) = s_{\text{head}}(p, f_a)^\frown s_{\text{tail}}(p, f_a) \leftarrow s_{\text{tail}}(p, f_a)$ (cf. lines 3–5).

- For each output port $p \in O$, the sequence of currently available tokens $s(p)$ is extended by the sequence of produced tokens $s_{\text{prod}}(p, f_a)$, i.e., $\forall p \in O : s(p) \leftarrow s(p)^\frown s_{\text{prod}}(p, f_a)$ (cf. lines 6–8).

- Finally, the actor state $s = (v, m_{\text{cur}})$ is updated with the corresponding transformed values v' and m' (cf. lines 9–10). Note that the selected target mode m' must be a valid possible target mode according to $t.M'$, i.e., $m' \in t.M'$. Otherwise, the model behavior is undefined.

Example 3.3. Consider again transition t_1 from Figure 3.2. Then, according to Definition 3.4, f_{a1} is defined as follows:

$$f_{a1} \colon \mathbb{Z}^{\text{cons}(i_1, f_{a1})} \times \mathbb{Z}^{\text{cons}(i_2, f_{a1})} \times \mathbb{Z}^{|v|} \times M \to \mathbb{Z}^{\text{prod}(o_1, f_{a1})} \times \mathbb{Z}^{|v|} \times M$$

Remember that $\text{cons}(i_1, f_{a1}) = 1$, $\text{cons}(i_2, f_{a1}) = 0$, and $\text{prod}(o_1, f_{a1}) = 0$ (cf. Example 3.1). Thus, the definition of f_{a1} can be rewritten as follows:

$$f_{a1} \colon \mathbb{Z}^1 \times \mathbb{Z}^0 \times \mathbb{Z}^6 \times M \to \mathbb{Z}^0 \times \mathbb{Z}^6 \times M$$

The behavior of f_{a1} consists in initializing the variable `mcuCount` to the value of the token s_1 consumed from input port i_1, setting the variable `i` to 0, and selecting m_2 from the set of possible target modes $t_1.M' = \{m_2\}$ (cf. Figure 2.5 on page 18):

$$f_{a1}(\langle s_1 \rangle, \langle\rangle, (\text{mcuCount}, \text{i}, \ldots, \text{tmp}), m_{\text{cur}}) \mapsto (\langle\rangle, (s_1, 0, \ldots, \text{tmp}), m_2)$$

The actor state $s = (v, m_{\text{cur}})$ and the token sequence $s(i_1)$ are subsequently updated accordingly.

Given a transition t, an important property of the guarded action semantics as described above is that if the guard function $t.f_g$ peeks at some tokens on an input port p, the action function $t.f_a$ must be provided the very same token sequence for which $t.f_g$ has been evaluated if t is subsequently executed. Otherwise, the model behavior is undefined.

Requirement 3.1. Given a transition t that is evaluated and subsequently executed. Then, for all input ports p, the sequence of tokens $s_{\text{head}}(p, t.f_g)$ evaluated by $t.f_g$ must be a *prefix*[3] of the sequence of tokens $s_{\text{head}}(p, t.f_a)$ supplied to $t.f_a$, i.e., $\forall p \in I : s_{\text{head}}(p, t.f_g) \sqsubseteq s_{\text{head}}(p, t.f_a)$.

Finally, transitions of the same actor are assumed to be *executed* sequentially. However, the execution of transitions may be pipelined, comparable to the pipelining of instructions which is typically performed by processors. As such optimizations are part of the micro-architecture generated by subsequent synthesis steps, they are not further discussed at this point. Note that transitions may be *evaluated* concurrently, as the evaluation of transitions is side-effect free. This approach cleanly decouples the dataflow concurrency model from the FSM semantics as proposed in [GLL99].

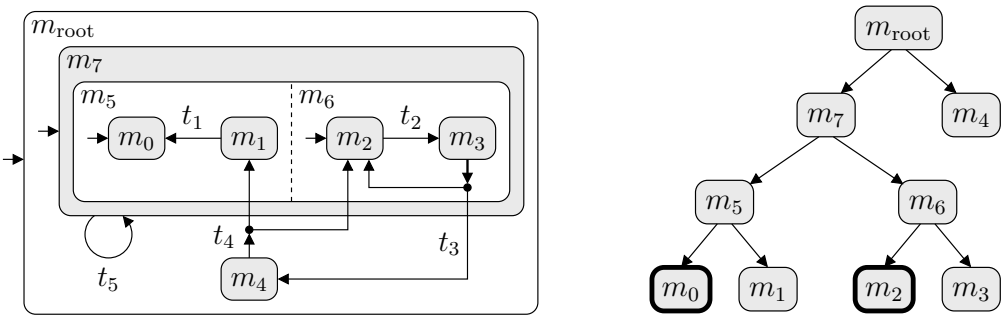
3.2 Hierarchical Modes

In order to cope with the complexity found in real-world applications, hierarchical FSMs are a solution. These are characterized by the fact that modes can contain other modes, which provides for a subsequent refinement of the behavior of an actor. Hierarchical modes have been used to model the running example from the networking domain in Chapter 6.

The basic model is extended by the two most important hierarchical modes known from Statecharts, namely AND modes and XOR modes [Har87]. In the case of XOR modes, the FSM is in exactly *one* of its child modes, whereas in the case of AND modes, the FSM is in *all* of its child modes. The key difference compared to [Har87] is that AND modes inherit the sequential execution semantics of transitions as described above. In particular, this means that transitions of different child modes of an AND mode are always executed sequentially. Thus, AND modes cannot be used to model the concurrent execution of transitions of an actor, but are solely used to reduce the number of modes and transitions of an actor compared to a functionally equivalent, non-hierarchical FSM.

In order to reflect the different kinds of modes, the set of actor modes M (cf. Definition 3.1) is partitioned into three disjoint subsets, namely $M = M_{\text{leaf}} \cup M_{\text{xor}} \cup M_{\text{and}}$. Here, M_{leaf} denotes the set of *leaf modes*, which correspond to the modes from the basic model without hierarchical modes. In contrast M_{xor} denotes the set of *XOR modes*, while M_{and} denotes the set of *AND modes*. We use the function $\mathrm{M}_{\text{child}} \colon M_{\text{xor}} \cup M_{\text{and}} \to \mathcal{P}(M)$ to denote the child modes of a hierarchical mode, and $\mathrm{M}_{\text{init}} \colon M_{\text{xor}} \to M$ to identify the user-defined initial child mode $\mathrm{M}_{\text{init}}(m) \in \mathrm{M}_{\text{child}}(m)$ of an XOR mode $m \in M_{\text{xor}}$.

[3] A sequence s_1 is a *prefix* of a sequence s_2, written $s_1 \sqsubseteq s_2$, if s_2 can be decomposed such that $s_2 = s_1 \frown s'$, where s' is a possibly empty sequence. Obviously, the empty sequence $\langle\rangle$ is a prefix of any other sequence s, i.e., $\langle\rangle \sqsubseteq s$.



a) Hierarchical modes b) Mode tree

Figure 3.3: a) Hierarchical modes example, and b) corresponding mode tree G_M induced by the parent/child relation of the modes. Modes m_0–m_4 are leaf modes, $m_5, m_6, m_{\text{root}}$ are XOR modes, and m_7 is the only AND mode. In order to visually convey this information, its child modes m_5 and m_6 are separated by dashed lines. The set of initial modes for the root mode is $\mathrm{M}_0(m_{\text{root}}) = \{m_0, m_2\}$.

The parent/child relation of modes induces a directed graph $G_M = (V_M, E_M)$, where the vertices V_M correspond to the modes M, and for each hierarchical mode $m \in M_{\text{xor}} \cup M_{\text{and}}$, an edge (m, m') is added to E_M for each child mode $m' \in \mathrm{M}_{\text{child}}(m)$. We require that the induced graph does not contain cycles, i.e., G_M must be a tree. Furthermore, we assume that a dedicated *root mode* $m_{\text{root}} \in M$ exists, such that all remaining modes $M' = M \setminus \{m_{\text{root}}\}$ are descendants of m_{root} w.r.t. G_M. In the following, we write $m_1 \preceq m_2$ if m_1 is an ancestor of m_2 w.r.t. G_M. Note that $m_1 \preceq m_2$ also if $m_1 = m_2$. Thus, "$\preceq$" defines a (non-strict) partial order over the set of modes M. The root mode m_{root} is an ancestor of every mode $m \in M$ (including m_{root}), i.e., $\forall m \in M : m_{\text{root}} \preceq m$.

Before discussing the operational semantics of transitions w.r.t. hierarchical modes, we refine the notion of the *current actor mode* and the *initial actor mode*. In principle, m_0 and m_{cur} are replaced by sets of leaf modes $\mathrm{M}_0 \subseteq M_{\text{leaf}}$ and $M_{\text{cur}} \subseteq M_{\text{leaf}}$, respectively. This is due to the fact that, for AND modes, all child modes are active simultaneously.

Thus, the set of initial modes for a given mode m, $\mathrm{M}_0 \colon M \to \mathcal{P}(M_{\text{leaf}})$ is recursively defined as follows:

$$\mathrm{M}_0(m) = \begin{cases} \{m\} & \text{if } m \in M_{\text{leaf}} \\ \mathrm{M}_0(\mathrm{M}_{\text{init}}(m)) & \text{if } m \in M_{\text{xor}} \\ \bigcup_{m_c \in \mathrm{M}_{\text{child}}(m)} \mathrm{M}_0(m_c) & \text{if } m \in M_{\text{and}} \end{cases} \tag{3.1}$$

For a leaf mode $m \in M_{\text{leaf}}$, the set of initial modes is defined to be only the mode m itself. For an XOR mode $m \in M_{\text{xor}}$, the set of initial modes is recursively computed from the user-defined initial mode $\mathrm{M}_{\text{init}}(m)$ of m. For an AND mode

$m \in M_{\text{and}}$, the set of initial modes is recursively computed from the initial modes of all child modes.

Initially, the set of current modes is $M_{\text{cur}} = \text{M}_0(m_{\text{root}})$. In the following, a mode $m \in M$ is said to be *active* if m is an ancestor mode of one of the current actor modes M_{cur}, i.e., m is active if $\exists m_{\text{cur}} \in M_{\text{cur}} : m \preceq m_{\text{cur}}$. In particular, as the root mode m_{root} is an ancestor of every mode, m_{root} is always active.

Example 3.4. Figure 3.3 gives an example for the proposed hierarchical modes. The overall set of modes is $M = \{m_0, \ldots, m_7, m_{\text{root}}\}$, with m_{root} being the root mode. The set of leaf modes consists of $M_{\text{leaf}} = \{m_0, \ldots, m_4\}$. The set of XOR modes consists of $M_{\text{xor}} = \{m_5, m_6, m_{\text{root}}\}$. The set of AND modes consists of $M_{\text{and}} = \{m_7\}$. The parent/child relation is indicated by the hierarchical structure of Figure 3.3a, and the corresponding mode tree G_M induced by the mode hierarchy is shown in Figure 3.3b. For example, the child modes of the AND mode m_7 consist of the XOR modes m_5 and m_6, i.e., $\text{M}_{\text{child}}(m_7) = \{m_5, m_6\}$.

According to Figure 3.3a, the initial modes of the XOR modes are defined as follows: $\text{M}_{\text{init}}(m_{\text{root}}) = m_7$, $\text{M}_{\text{init}}(m_5) = m_0$, and $\text{M}_{\text{init}}(m_6) = m_2$. Thus, the set of initial modes of the root mode m_{root} can be calculated according to Equation (3.1) as $\text{M}_0(m_{\text{root}}) = \{m_0, m_2\}$. This means that, initially, $M_{\text{cur}} = \{m_0, m_2\}$, and hence, all modes except m_1, m_3 and m_4 are active.

Intuitively, not all possible combinations of leaf modes result in a valid set of current modes. Let $\text{M}_{\text{active}}(m) = \{m' \in \text{M}_{\text{child}}(m) \mid \exists m_{\text{cur}} \in M_{\text{cur}} : m' \preceq m_{\text{cur}}\}$ be the set of active child modes of a hierarchical mode $m \in M_{\text{xor}} \cup M_{\text{and}}$.

Definition 3.6 (Valid set of current modes)**.** A set of current modes $M_{\text{cur}} \subseteq M_{\text{leaf}}$ is *valid* if the following conditions hold for all hierarchical modes $m \in M_{\text{xor}} \cup M_{\text{and}}$:

- If $m \in M_{\text{xor}}$ (i.e., m is an XOR mode), at most one of its child modes must be active: $|\text{M}_{\text{active}}(m)| \leq 1$. Note that if none of its child modes are active, m is not active, which is perfectly valid.
- If $m \in M_{\text{and}}$ (i.e, m is an AND mode), either none of its child modes must be active, or all of its child modes must be active: $|\text{M}_{\text{active}}(m)| = 0 \vee |\text{M}_{\text{active}}(m)| = |\text{M}_{\text{child}}(m)|$. Again, if none of its child modes are active, m is not active, which is also perfectly valid in this case.

Example 3.5. Consider again Figure 3.3. Initially, $M_{\text{cur}} = \{m_0, m_2\}$, which represents a valid set of leaf modes: $\text{M}_{\text{active}}(m_5) = \{m_0\}$, $\text{M}_{\text{active}}(m_6) = \{m_2\}$, $\text{M}_{\text{active}}(m_7) = \{m_5, m_6\}$, and $\text{M}_{\text{active}}(m_{\text{root}}) = \{m_7\}$. In contrast, $M_{\text{cur}} = \{m_1\}$ would be an invalid set of modes, because $|\text{M}_{\text{active}}(m_7)| = |\{m_5\}| = 1 \neq 2 = |\text{M}_{\text{child}}(m_7)|$.

Transitions are not modified compared to the basic model, i.e., a transition t specifies a single source mode $m \in M$ in which it is active, and a set of possible target modes $M' \subseteq M$. The operational semantics of the hierarchical FSM are defined in terms of the non-hierarchical FSM from Definition 3.1. To this end, the hierarchical FSM must be *flattened*. This can be achieved, for example, by a symbolic execution of the hierarchical FSM which computes all reachable sets of modes, starting from the initial set of modes $\mathrm{M}_0(m_{\mathrm{root}})$. This process is detailed in the following section.

3.2.1 Operational Semantics

Starting with a valid set of current modes $M_{\mathrm{cur}} = \mathrm{M}_0(m_{\mathrm{root}})$ according to Definition 3.6, each transition $t \in T$ is analyzed whether it is active according to the set of current modes M_{cur} or not. Note that the transitions can be processed in isolation, as AND modes inherit the sequential execution semantics of the basic model, i.e., transitions of different child modes of an AND mode are always executed sequentially. A transition t is active if its source mode $t.m$ is active, i.e., if $\exists m_{\mathrm{cur}} \in M_{\mathrm{cur}} : t.m \preceq m_{\mathrm{cur}}$. If t is active according to M_{cur}, for each possible target mode $m' \in t.M'$, the set of target modes in the hierarchical model is calculated based on the set of current modes M_{cur}.

Executing an active (and enabled) transition t means transitioning the FSM from mode $t.m$ to a mode $m' \in t.M'$. In the following, m denotes $t.m$, while m' denotes a possible target mode $m' \in t.M'$. Then, transitioning the FSM from a mode m to a mode m' corresponds to a traversal of the mode tree G_M induced by the mode hierarchy: Starting from the source mode m, G_M is traversed *upwards* until the lowest common ancestor (LCA) mode $\hat{m}$ of m and m' is reached. The LCA mode $\hat{m} = \mathrm{lca}(m_1, m_2)$ of two modes m_1 and m_2 is a mode $\hat{m}$ such that $\hat{m} \preceq m_1 \wedge \hat{m} \preceq m_2 \wedge \nexists \hat{m}' \neq \hat{m} : \hat{m}' \preceq m_1 \wedge \hat{m}' \preceq m_2 \wedge \hat{m} \preceq \hat{m}'$. Note that in case of the rooted mode tree G_M, such an LCA mode can always be found for two modes. The modes encountered during this upward traversal are *left* (i.e., become inactive), including the LCA mode $\hat{m}$. Then, starting from the LCA mode $\hat{m}$, G_M is traversed *downwards* until the target mode m' is reached. The modes encountered during this downward traversal are *entered* (i.e., become active), including the LCA mode $\hat{m}$. The resulting set of active leaf modes M'_{cur} then becomes the new set of current modes M_{cur}.

Example 3.6. Consider again Figure 3.3. Given the initial set of current modes $M_{\mathrm{cur}} = \{m_0, m_2\}$, only transitions t_2 and t_5 are active: For t_2, we have that $t_2.m = m_2 \preceq m_2 \in M_{\mathrm{cur}}$, and for t_5, we have that $t_5.m = m_7 \preceq m_0 \in M_{\mathrm{cur}}$. If t_2 is executed, the FSM transitions from mode m_2 to mode m_3. It follows that $\hat{m} = \mathrm{lca}(m_2, m_3) = m_6$ (cf. Figure 3.3b). The tree traversal results in the

Algorithm 3.3 Selection of child modes

```
1: procedure M_select(m ∈ M_xor, m' ∈ M)
2:     if ∃m_c ∈ M_child(m) : m_c ⪯ m' then
3:         return m_c
4:     else
5:         return M_init(m)
6:     end if
7: end procedure
```

following operations: (1) leave m_2, (2) leave m_6, (3) re-enter m_6, and (4) enter m_3. The new set of current modes is therefore $M'_{\text{cur}} = \{m_0, m_3\}$.

If t_5 is executed, the FSM transitions from mode m_7 to mode m_7. In this case, it follows that $\hat{m} = \text{lca}(m_7, m_7) = m_7$. The tree traversal results in the following operations: (1) leave m_7, and (2) re-enter m_7. The new set of current modes is therefore $M'_{\text{cur}} = \{m_0, m_2\}$.

We still have to formalize the concepts of *leaving* and *entering* a mode m. Leaving a mode means that all leaf modes which are descendants of m become inactive. More formally, function leave calculates this set of leaf modes for a given mode m:

$$\text{leave}(m) = \{m_l \in M_{\text{leaf}} \mid m \preceq m_l\} \tag{3.2}$$

The entering of a mode m is more complicated, as the semantics depend on the kind of mode that is entered and the target mode m'. To this end, the procedure to determine the set of initial modes of a mode (cf. Equation (3.1)) is modified such that when entering an XOR mode m, the initial mode $\text{M}_{\text{init}}(m) \in \text{M}_{\text{child}}(m)$ is only entered (i.e., becomes active) if no other child mode $m_c \in \text{M}_{\text{child}}(m)$ exists which is an ancestor mode of the target mode m'. If such a child mode m_c exists, m_c must be entered instead of the user-defined initial mode $\text{M}_{\text{init}}(m)$. This mode selection process is summarized by Algorithm 3.3. Function enter is then defined as follows:

$$\text{enter}(m, m') = \begin{cases} \{m\} & \text{if } m \in M_{\text{leaf}} \\ \text{enter}(\text{M}_{\text{select}}(m, m'), m') & \text{if } m \in M_{\text{xor}} \\ \bigcup_{m_c \in \text{M}_{\text{child}}(m)} \text{enter}(m_c, m') & \text{if } m \in M_{\text{and}} \end{cases} \tag{3.3}$$

Given a set of current modes M_{cur}, a source mode m, a target mode m', and the LCA mode $\hat{m}$ of m and m', the new set of current modes M'_{cur} can then be calculated as follows:

$$M'_{\text{cur}} = (M_{\text{cur}} \setminus \text{leave}(\hat{m})) \cup \text{enter}(\hat{m}, m') \tag{3.4}$$

Example 3.7. Consider again Figure 3.3. Given the initial set of current modes $M_{\text{cur}} = \{m_0, m_2\}$, only transitions t_2 and t_5 are active (cf. Example 3.6). If t_2 is executed, the FSM transitions from mode m_2 to mode m_3. In this case, it follows that $\hat{m} = \text{lca}(m_2, m_3) = m_6$ (cf. Figure 3.3b). Thus, $\text{leave}(m_6) = \{m_2, m_3\}$, and $\text{enter}(m_6, m_3) = \{m_3\}$ (because $\text{M}_{\text{select}}(m_6, m_3) = m_3$). Thus, $M'_{\text{cur}} = (\{m_0, m_2\} \setminus \{m_2, m_3\}) \cup \{m_3\} = \{m_0, m_3\}$.

If t_5 is executed, the FSM transitions from mode m_7 to mode m_7. In this case, it follows that $\hat{m} = \text{lca}(m_7, m_7) = m_7$. Thus, $\text{leave}(m_7) = \{m_0, m_1, m_2, m_3\}$, and $\text{enter}(m_7, m_7) = \{m_0, m_2\}$. Thus, $M'_{\text{cur}} = (\{m_0, m_2\} \setminus \{m_0, m_1, m_2, m_3\}) \cup \{m_0, m_2\} = \{m_0, m_2\} = M_{\text{cur}}$.

Theorem 3.1. Given a source mode $m \in M$, a target mode $m' \in M$, and a valid set of current modes M_{cur} according to Definition 3.6. Then, the set of target modes M'_{cur} computed by Equation (3.4) is also a valid set of modes according to Definition 3.6.

Proof. Let $\hat{m} = \text{lca}(m, m')$. If a mode $m_e \in M$ is entered, it follows that $\hat{m} \preceq m_e$. However, $\text{leave}(\hat{m})$ leaves all child modes which are descendants of $\hat{m}$. As m_e is a descendant of $\hat{m}$, it directly follows that, in particular, all child modes which are descendants of m_e are left by $\text{leave}(\hat{m})$. Thus, if m_e is entered, it cannot be active anymore. To complete the proof, we have to show that enter as per Equation (3.3) is correct according to Definition 3.6. However, this is trivially accomplished: for a leaf mode $m_e \in M_{\text{leaf}}$, only m_e itself is entered. For an XOR mode $m_e \in M_{\text{xor}}$, exactly one mode according to $\text{M}_{\text{select}}(m_e, m')$ is entered. Finally, for an AND mode $m_e \in M_{\text{and}}$, all child modes are entered. This behavior adheres to Definition 3.6. □

An immediate consequence of Theorem 3.1 is that all combinations of source and target modes are valid for a transition t. The FSM flattening procedure is summarized by Algorithm 3.4. Note that each mode $\overline{m} \in \overline{M}$ of the flattened FSM corresponds to a set of leaf modes of the hierarchical FSM. At a glance, the algorithm selects an unprocessed set of modes as the current set of modes M_{cur}, and subsequently determines the transitions which are active in M_{cur} (cf. lines 6–8). Then, for each active transition, the set of possible target modes $\overline{M'}$ of the transition w.r.t. the flattened FSM are computed (cf. lines 9–14). Subsequently, a transition $\overline{t}$ is added to the set of transitions $\overline{T}$ of the flattened FSM (cf. lines 15–16). Finally, the set of possible target modes $\overline{M'}$ is added to the set of modes $\overline{M}$ of the flattened FSM (cf. line 17). When all transitions which are active in M_{cur} have been processed, the current set of modes M_{cur} is marked as processed (cf. line 20). When no unprocessed set of modes remains, the flattening process is finished, and the flattened FSM is returned (cf. line 22)

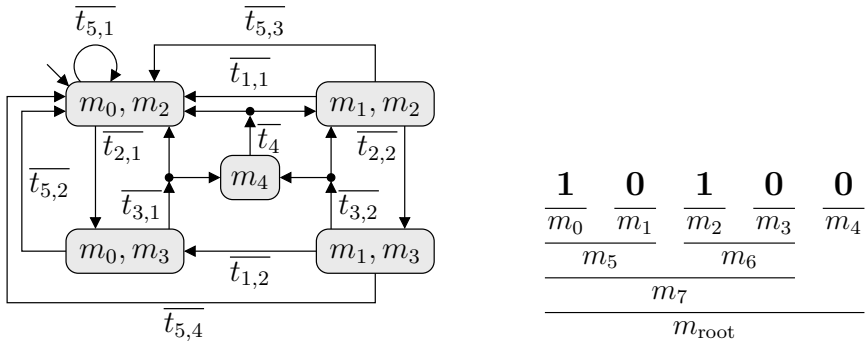
Algorithm 3.4 Flattening of hierarchical FSMs

procedure FLATTENFSM(Modes M, Root mode $m_{\text{root}} \in M$, Transitions T)
  Let $\overline{m_0} \leftarrow M_0(m_{\text{root}})$ ▷ Initial mode of the flattened FSM
  Let $\overline{M} \leftarrow \{\overline{m_0}\}$ ▷ Modes of the flattened FSM
  Let $\overline{T} \leftarrow \emptyset$ ▷ Transitions of the flattened FSM
  Let $M_p \leftarrow \emptyset$ ▷ Processed sets of modes
  for all $M_{\text{cur}} \in \overline{M} \setminus M_p$ **do**
    for all $t \in T$ **do**
      if $\exists m_{\text{cur}} \in M_{\text{cur}} : t.m \preceq m_{\text{cur}}$ **then**
        Let $\overline{M'} \leftarrow \emptyset$
        for all $m' \in t.M'$ **do**
          Let $\hat{m} \leftarrow \text{lca}(t.m, m')$
          Let $\overline{m'} \leftarrow (M_{\text{cur}} \setminus \text{leave}(\hat{m})) \cup \text{enter}(\hat{m}, m')$
          $\overline{M'} \leftarrow \overline{M'} \cup \{\overline{m'}\}$
        end for
        Let $\bar{t} = (M_{\text{cur}}, \overline{M'}, t.f_g, t.f_a)$
        $\overline{T} \leftarrow \overline{T} \cup \{\bar{t}\}$
        $\overline{M} \leftarrow \overline{M} \cup \overline{M'}$
      end if
    end for
    $M_p \leftarrow M_p \cup \{M_{\text{cur}}\}$
  end for
  return $(\overline{M}, \overline{m_0}, \overline{T})$
end procedure

Example 3.8. The flattened FSM corresponding to the hierarchical FSM from Figure 3.3 is shown in Figure 3.4a. The initial mode $\overline{m_0} = \{m_0, m_2\} = M_0(m_{\text{root}})$ corresponds to the initial set of modes for the root mode of the hierarchical FSM. While the number of modes of the flattened FSM is equal to the number of leaf modes of the hierarchical FSM, the former has more than twice as many transitions as the latter.

3.2.2 Implementation

While the flattening of a hierarchical FSM is useful for analytic purposes, an implementation may abstain from doing so, because the number of modes and transitions may become very large. A more efficient solution consists in keeping the hierarchical FSM. To this end, a bit vector of size $|M_{\text{leaf}}|$ can be allocated, where a bit b_i corresponds to a leaf mode $m_i \in M_{\text{leaf}}$. Bit b_i is set only if m_i is active, i.e., if $m_i \in M_{\text{cur}}$.



a) Flattened FSM b) Bit vector

Figure 3.4: a) Flattened FSM corresponding to the hierarchical FSM from Figure 3.3, and b) bit vector reflecting the leaf modes from Figure 3.3.

Example 3.9. A possible bit vector for the leaf modes of the hierarchical FSM from Figure 3.3 is shown in Figure 3.4b. As the initial set of current modes is $M_{\text{cur}} = M_0(m_{\text{root}}) = \{m_0, m_2\}$, only the bits corresponding to leaf modes m_0 and m_2 are set initially.

Given such a bit vector, the following operations can be efficiently implemented:

- In order to determine whether a mode $m \in M$ is active, at least one of the bits corresponding to m must be set.
- In order to leave a mode m, the bits corresponding to leave(m) must be cleared.
- In order to enter a mode m, the bits corresponding to enter(m, m') must be set (m' is the target mode).

As the functions enter and leave only depend on the source mode and possible target modes of a transition and not on the set of current modes, the corresponding bit masks can be precomputed at compile time.

Example 3.10. Consider again Figure 3.3 and Figure 3.4b. All modes except m_4 are active. For example, the bits corresponding to m_6 are $(1, 0) \neq \mathbf{0}$. The bits corresponding to m_7 are $(1, 0, 1, 0) \neq \mathbf{0}$.

Given the initial set of current modes $M_{\text{cur}} = \{m_0, m_2\}$, only transitions t_2 and t_5 are active (cf. Example 3.6). If t_2 is executed, the FSM transitions from mode m_2 to mode m_3. As $\hat{m} = \text{lca}(m_2, m_3) = m_6$, it follows that leave($m_6$) = $\{m_2, m_3\}$ and enter(m_6, m_3) = $\{m_3\}$ (cf. Example 3.7). Concerning the bit

vector, leave(m_6) clears the corresponding bits, resulting in a new bit vector $(1, 0, 0, 0, 0)$. Subsequently, enter(m_6, m_3) sets the corresponding bits, resulting in a new bit vector $(1, 0, 0, 1, 0)$.

3.2.3 Extensions

In [ZFH+10], hierarchical transitions have been extended compared to transitions of the basic model. In particular, a transition t may specify a set of modes M^+ (in addition to $t.m$) which must also be active, and a set of modes M^- which must not be active. This behavior can be easily achieved by extending the notion of an active mode.

Furthermore, a transition t may specify a set of modes which are entered when t is executed (not to be confused with the set of possible target modes $t.M'$, of which exactly one mode is entered when t is executed). This behavior can be easily achieved by extending the function enter to accommodate a set of target modes.

3.3 Restricted Model

In this section, a restricted actor model is described which does not allow the use of guard functions. It can be observed that this restricted model may lead to more accurate analysis results and moreover simplifies some synthesis tasks as outlined in the corresponding sections. For example, in the context of hardware synthesis, some data hazards cannot occur in the absence of guard functions. In the following, it is shown that the restricted model is as expressive as the basic actor model. To this end, the basic model is transformed into the restricted model. However, in the worst case, the number of modes may grow exponentially due to this transformation. As result, we can conclude that certain problems can be represented much more efficiently (i.e., by smaller FSMs) by the use of guard functions. Moreover, guard functions are side-effect free by definition, which allows them to be evaluated in parallel in principle. This parallelism is lost in the restricted model.

The restricted model is derived from the basic model by eliminating the set of guard functions F_g and the function peek from Definition 3.1, and the guard function f_g from Definition 3.5. Concerning the operational semantics, line 11 of Algorithm 3.1 is replaced by "return $\top$". Note that the process transformation as outlined in Section 2.3 generates transitions which adhere to the restricted actor model, i.e., they don't require guard functions.

In order to transform an FSM with guard functions into an equivalent FSM without guard functions, one possibility is to perform a symbolic evaluation,

like done for the transformation of hierarchical FSMs into the basic model (cf. Algorithm 3.4). To this end, starting from the initial mode m_0, all outgoing transitions are transformed into transitions $t' = (m, M', f_a)$ according to the restricted model as explained in the following.

First, each transition $t = (m, M', f_g, f_a)$ is split into two partial transitions, namely $t'_{\text{grd}} = (m, \{m_\top, m_\bot\}, f'_g)$ and $t'_{\text{act}} = (m_\top, M', f'_a)$. Basically, the action function f'_g of t'_{grd} evaluates the guard function f_g, while the action function f'_a of t'_{act} executes the action function f_a. Note that t_i must be split in the general case because tokens may be produced by f_a, i.e., if $\exists p \in O : \text{cons}(p, f_a) > 0$. In this case, the tokens can only be produced if f_g evaluates to $\top$.

When t'_{grd} is executed, its action function f'_g evaluates the guard function f_g. For the sake of readability, we assume that t'_{grd} consumes all tokens according to the values of cons, and not just those specified by peek. As the result of f_g cannot be queried by guard functions in the restricted model, it is captured by two additional modes $m_\top$ and $m_\bot$ which are the possible target modes of t'_{grd}. Additionally, f'_g caches the tokens consumed from an input port $p \in I$ in additional actor variables. This corresponds to the semantics of the basic model which requires that guard functions do not consume tokens: The cached tokens are valid until a partial transition t'_{act} is executed which "consumes" (i.e., invalidates) some (or all) of these cached tokens. In the following, we use the function $\text{c}_{\text{valid}} \colon I \times M \to \mathbb{N}_0$ to denote the number of valid cached tokens in a mode m for an input port $p \in I$.

Then, for the partial transition t'_{grd} attached to mode m, it follows that $\forall p \in O : \text{prod}(p, f'_g) = 0$, and $\forall p \in I : \text{cons}(p, f'_g) = \max\{0, \text{cons}(p, f_a) - \text{c}_{\text{valid}}(p, m)\}$. Note that if enough tokens are cached, no tokens have to be consumed at all. For the possible target modes $m_\top$ and $m_\bot$ of t'_{grd}, it follows that $\forall p \in I : \text{c}_{\text{valid}}(p, m_\top) = \text{c}_{\text{valid}}(p, m_\bot) = \text{c}_{\text{valid}}(p, m) + \text{cons}(p, f'_g) = \text{c}_{\text{valid}}(p, m) + \max\{0, \text{cons}(p, f_a) - \text{c}_{\text{valid}}(p, m)\} = \max\{\text{c}_{\text{valid}}(p, m), \text{cons}(p, f_a)\}$. The maximal number of tokens cached for an input port p can be statically determined to $\text{c}_{\max}(p) = \max_{f_a \in F_a}\{\text{cons}(p, f_a)\}$. In the following, $v_{\text{cache}}(p)$ denotes the $\text{c}_{\max}(p)$-tuple of actor variables used to cache the tokens consumed from an input port $p \in I$. In addition, v_{orig} denotes the original actor variables without the added variables used as token cache.

The partial transition t'_{act} is only active in mode $m_\top$, and waits for free places. Note that all tokens required by t'_{act} have already been consumed (and cached) by t'_{grd}. When executed, its action function f'_a executes the action function f_a. Thus, for the partial transition t'_{act} attached to mode $m_\top$, it follows that $\forall p \in O : \text{prod}(p, f'_a) = \text{prod}(p, f_a)$, and $\forall p \in I : \text{cons}(p, f'_a) = 0$. Furthermore, for the possible target modes M' of t'_{act}, it follows that $\forall m' \in M' : \forall p \in I : \text{c}_{\text{valid}}(p, m') = \text{c}_{\text{valid}}(p, m_\top) - \text{cons}(p, f_a)$.

Note that the mode $m_\bot$ has no outgoing transitions (for now), as t'_{act} is only attached to $m_\top$: If the original transition t is the only transition attached to its

Algorithm 3.5 Behavior of the action function f'_g

1: **procedure** $f'_g(s_{\text{head}}(i_1, f'_g), \ldots, s_{\text{head}}(i_{|I|}, f'_g), v, m)$
2: Let $v_{\text{orig}}{}^\frown v_{\text{cache}}(i_1)^\frown \ldots {}^\frown v_{\text{cache}}(i_{|I|}) \leftarrow v$
3: **for all** $p \in I$ **do**
4: Let $v_{\text{valid}}(p)^\frown v_{\text{update}}(p)^\frown v_{\text{invalid}}(p) \leftarrow v_{\text{cache}}(p)$
s.t. $|v_{\text{valid}}(p)| = \text{c}_{\text{valid}}(p, m)$ and $|v_{\text{update}}(p)| = \left|s_{\text{head}}(p, f'_g)\right|$
5: Let $v'_{\text{cache}}(p) \leftarrow v_{\text{valid}}(p)^\frown s_{\text{head}}(p, f'_g)^\frown v_{\text{invalid}}(p)$
6: Let $v_{\text{head}}(p)^\frown v_{\text{tail}}(p) \leftarrow v'_{\text{cache}}(p)$ s.t. $|v_{\text{head}}(p)| = \text{peek}(p, f_g)$
7: **end for**
8: **if** $f_g(v_{\text{head}}(i_1), \ldots, v_{\text{head}}(i_{|I|}), v_{\text{orig}}, m) = \top$ **then**
9: Let $m' \leftarrow m_\top$
10: **else**
11: Let $m' \leftarrow m_\bot$
12: **end if**
13: Let $v' \leftarrow v_{\text{orig}}{}^\frown v'_{\text{cache}}(i_1)^\frown \ldots {}^\frown v'_{\text{cache}}(i_{|I|})$
14: **return** $(\langle\rangle, \ldots, \langle\rangle, v', m')$
15: **end procedure**

source mode $t.m$, it can be observed that the original FSM would be stuck in mode $t.m$ if the guard function $t.f_g$ evaluates to $\bot$: Once evaluated, the result of $t.f_g$ can only change if some of the tokens accessed by $t.f_g$ are consumed, or if the actor variables read by $t.f_g$ are modified. Both conditions, however, can only be met if a transition of the actor in question is executed. (Remember that we assume channels to be point-to-point connections.) However, as t is the only transition attached to its source mode $t.m$, and $t.f_g$ is assumed to evaluate to $\bot$, no other transition can be executed for the actor in question. Consequently, the transformed FSM would be stuck in mode $m_\bot$ in this case. If the mode $t.m$ has some outgoing transitions besides t, the partial transitions of all outgoing transitions of $t.m$ will be *interleaved*. Then, $m_\bot$ has some outgoing transitions in principle. This interleaving of partial transitions is described later.

Algorithm 3.5 summarizes the behavior of f'_g: First, for each input port $p \in I$, $v'_{\text{cache}}(p)$ is computed from $v_{\text{cache}}(p)$ by replacing the appropriate token subsequence by the tokens consumed from p (i.e., $s_{\text{head}}(p, f'_g)$) (cf. lines 3–7). Note that in particular, cached tokens which are already valid are not overwritten. Subsequently, the guard function f_g is evaluated, which determines the target mode of the transition (cf. lines 8–12). To this end, the cached tokens are supplied, as well as the original actor variables. Finally, the modified actor variables and the selected target mode are returned by f'_g (cf. line 14). Note that f'_g does not produce any tokens, i.e., $\forall p \in O : \text{cons}(p, f'_g) = 0$, corresponding to the empty sequences in line 14.

Algorithm 3.6 Behavior of the action function f'_a

1: **procedure** $f'_a(s_{\text{head}}(i_1, f'_a), \ldots, s_{\text{head}}(i_{|I|}, f'_a), v, m)$
2: Let $v_{\text{orig}}{}^\frown v_{\text{cache}}(i_1)^\frown \ldots {}^\frown v_{\text{cache}}(i_{|I|}) \leftarrow v$
3: **for all** $p \in I$ **do**
4: Let $v_{\text{head}}(p)^\frown v_{\text{tail}}(p) \leftarrow v_{\text{cache}}(p)$ s.t. $|v_{\text{head}}(p)| = \text{cons}(p, f_a)$
5: Let $v'_{\text{cache}}(p) \leftarrow v_{\text{tail}}(p)^\frown v_{\text{head}}(p)$
6: **end for**
7: Let $(s_{\text{prod}}(o_1, f'_a), \ldots, s_{\text{prod}}(o_{|O|}, f'_a), v'_o, m') \leftarrow$
$f_a(v_{\text{head}}(i_1), \ldots, v_{\text{head}}(i_{|I|}), v_{\text{orig}}, m)$
8: Let $v' \leftarrow v'_o{}^\frown v'_{\text{cache}}(i_1)^\frown \ldots {}^\frown v'_{\text{cache}}(i_{|I|})$
9: **return** $(s_{\text{prod}}(o_1, f'_a), \ldots, s_{\text{prod}}(o_{|O|}, f'_a), v', m')$
10: **end procedure**

Algorithm 3.6 summarizes the behavior of f'_a: First, for each input port $p \in I$, $v'_{\text{cache}}(p)$ is computed from $v_{\text{cache}}(p)$ by invalidating the tokens consumed by f_a (cf. lines 3–6). Subsequently, the action function f_a is executed (cf. line 7). To this end, the cached tokens are supplied, as well as the original actor variables. Subsequently, the produced tokens, the modified actor variables, and the selected target mode are returned by f'_a (cf. line 9).

Example 3.11. Consider the FSM shown in Figure 3.5a. It consists of the single mode m_0 and two transitions t_1 and t_2, both of which are attached to m_0. Initially, splitting t_1 results in $t'_1 = (m_0, \{m_{1\top}, m_{1\bot}\}, f'_{g1})$ and $t'_2 = (m_{1\top}, m_0, f'_{a1})$, while splitting t_2 results in $t'_3 = (m_0, \{m_{2\top}, m_{2\bot}\}, f'_{g2})$ and $t'_4 = (m_{2\top}, m_0, f'_{a2})$. While f'_{g1} evaluates f_{g1} as described in Algorithm 3.5, f'_{a1} executes f_{a1} as described in Algorithm 3.6. Analogously, f'_{g2} evaluates f_{g2}, and f'_{a2} executes f_{a2}. The number of valid cached tokens is annotated as tuple $(\text{c}_{\text{valid}}(i_1, m), \text{c}_{\text{valid}}(i_2, m))$ to each mode m. For example, in mode $m_{1\top}$, one token from i_1 is cached, while for i_2, no tokens are cached. In contrast, in mode $m_{2\top}$, no tokens from i_1 are cached, while for i_2, one token is cached.

Having determined the partial transitions, the next step of the transformation process consists in interleaving the partial transitions of all outgoing transitions of a mode m. While only described informally, this step is necessary due to the additional modes $m_\top$ and $m_\bot$ which have been introduced to capture the result of a guard function: On the one hand, if the FSM transitions into $m_\top$ by means of $t'_{i,\text{grd}}$, and if $t'_{i,\text{act}}$ is the only outgoing transition of $m_\top$, the FSM would remain in $m_\top$ until enough free places are available in order to execute f'_a, which may never be the case, depending on the behavior of the environment. On the other hand, if the FSM transitions into $m_\bot$, no outgoing transitions means that the FSM would be stuck in $m_\bot$ indefinitely. Both cases show that

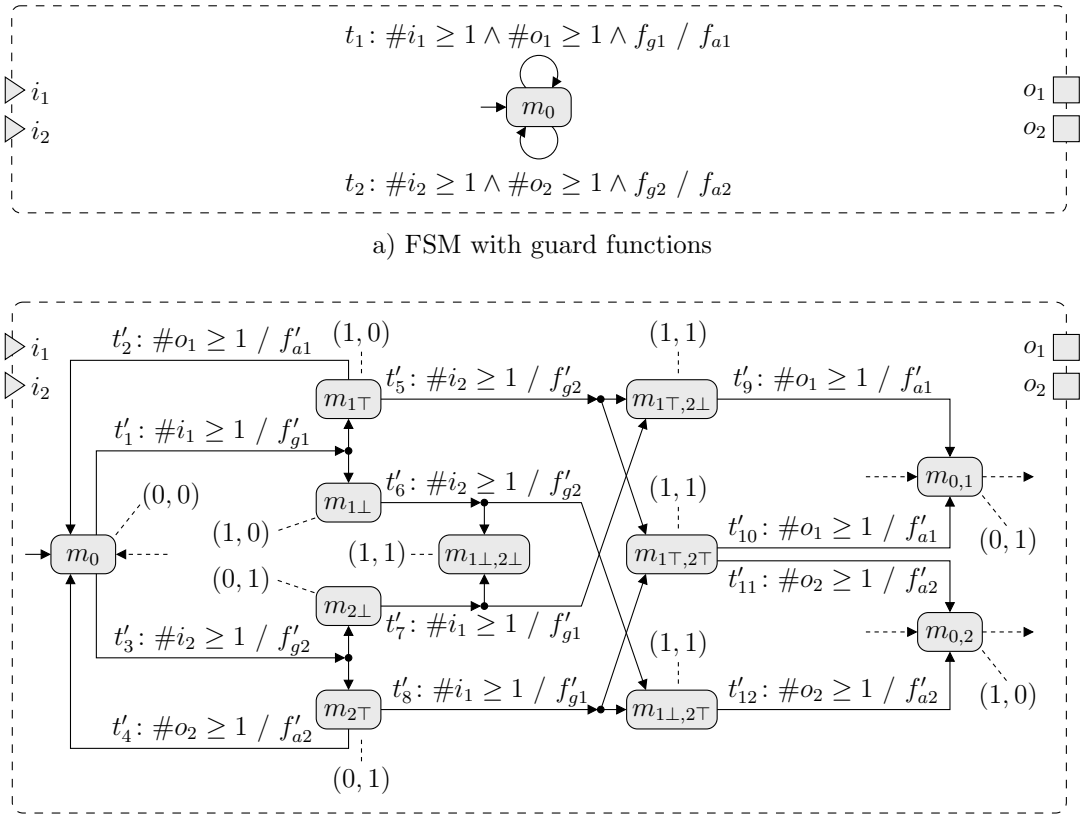


Figure 3.5: Transformation of an FSM with guard functions (a) into an equivalent FSM without guard functions (b). The tuple annotated to a mode m specifies the values of $\mathrm{c}_{\mathrm{valid}}(i_1, m)$ and $\mathrm{c}_{\mathrm{valid}}(i_2, m)$.

the partial transition $t'_{j,\mathrm{grd}}$ of another transition $t_j \neq t_i$ with $t_j.m = t_i.m$ must be attached to both modes $m_\top$ and $m_\bot$, which, of course, requires duplicating $t'_{j,\mathrm{grd}}$ appropriately. The interleaving process results in new modes which encode all possible combinations of guard results. For each of these modes m, the number of valid cached tokens $\mathrm{c}_{\mathrm{valid}}(p, m)$ corresponds to the maximal number of tokens consumed from each input port by the partial transitions $t'_{i,\mathrm{grd}}$ and $t'_{j,\mathrm{grd}}$.

Example 3.12. Consider again the FSM shown in Figure 3.5a. Interleaving the partial transitions corresponding to transitions t_1 and t_2 leads to the four additional modes which encode the results of both f_{g1} and f_{g2}. For example, both modes $m_{1\top}$ and $m_{1\bot}$ now have outgoing transitions, namely t'_5 and t'_6, which correspond to the partial transition t'_3. Analogously, both modes $m_{2\top}$ and $m_{2\bot}$ now have outgoing transitions, namely t'_7 and t'_8, which correspond to the partial transition t'_1 (cf. Example 3.11). However, if both guard functions

f_{g1} and f_{g2} have been evaluated to false, the FSM transitions into mode $m_{1\perp,2\perp}$ which has no outgoing transitions. Note that in all four additional modes, one token is cached for each input port i_1 and i_2.

While interleaving the partial transitions is necessary in order to not introduce deadlocks into the model, it also creates a new problem: There may still exist valid cached tokens even after a partial transition t'_{act} has been executed. Given two transitions t_i and t_j, $t_i \neq t_j$. Assume that first, the partial transition $t'_{i,\text{grd}}$ is executed, then $t'_{j,\text{grd}}$, and finally $t'_{j,\text{act}}$. This means that $t'_{i,\text{grd}}$ may consume (and cache) tokens which are not required by $t'_{j,\text{grd}}$. In turn, these cached tokens are still valid after $t'_{j,\text{act}}$ has been executed.

Let $m' \in t.M'$ be a possible target mode of a transition t for which the partial transition t'_{act} has been executed. Assume that a token cache configuration is already associated with m' where no cached tokens are valid, i.e., $\forall p \in I : \mathrm{c}_{\text{valid}}(p, m') = 0$. If after executing t'_{act}, however, some cached tokens are still valid, m' is not a feasible target mode, as the calculated token cache configuration conflicts with the values of $\mathrm{c}_{\text{valid}}$ for mode m'. In this case, a new mode m'_k must be allocated corresponding to m', and the possible set of target modes of t'_{act} is set to $t'_{\text{act}}.M' = (t.M' \setminus \{m'\}) \cup \{m'_k\}$. Note that the outgoing transitions of m' are duplicated and attached to m'_k accordingly. Subsequently, the calculated token cache configuration is associated with m'_k. Depending on the interleaving of partial transitions, multiple such modes may be allocated for a mode m', each of which corresponds to a different token cache configuration.

The transformation process then continues with m'_k. The transformation process is finished when a fixed point is reached, i.e., no new modes are added to the FSM according to the restricted model.

Example 3.13. Consider again the FSM shown in Figure 3.5b. Assume that t'_1 is executed first, followed by t'_5, and finally, t'_{11}. In this case, the token consumed from i_1 which is cached by t'_1 is still valid after both t'_5 and t'_{11} have been executed (which corresponds to transition t_2 in Figure 3.5a). In contrast, the token consumed from i_2 which is cached by t'_5 is no longer valid. Thus, mode $m_{0,2}$ encodes $\mathrm{c}_{\text{valid}}(m_{0,2}, i_1) = 1$ and $\mathrm{c}_{\text{valid}}(m_{0,2}, i_2) = 0$. Starting from mode $m_{0,2}$, the transformation procedure basically duplicates the FSM shown in Figure 3.5b, but for i_1, reduces the values of cons accordingly (not shown). Analogously, mode $m_{0,1}$ encodes $\mathrm{c}_{\text{valid}}(m_{0,1}, i_1) = 0$ and $\mathrm{c}_{\text{valid}}(m_{0,1}, i_2) = 1$, leading again to a duplication of the FSM from Figure 3.5b with reduced values of cons for i_2. Note that for mode m_0, $\mathrm{c}_{\text{valid}}(m_0, i_1) = 0$ and $\mathrm{c}_{\text{valid}}(m_0, i_2) = 0$.

Let $T(m) = \{t \in T \mid t.m = m\}$ be the set of outgoing transitions of a mode $m \in M$, and $n = |T(m)| \geq 1$. For the worst case interleaving of split transitions

resulting from $T(m)$, it can be shown that the number of modes $M_{\text{guard}}(m)$ which encode the results of guard functions amounts to:

$$|M_{\text{guard}}(m)| = \sum_{k=1}^{n} \binom{n}{k} 2^k = 3^n - 1 = 3^{|T(m)|} - 1 \tag{3.5}$$

Thus, the number of modes $M_{\text{guard}}(m)$ grows exponentially with the number of outgoing transitions of a mode m. The transformation procedure shows that while it is possible to cope without guard functions, certain problems can be represented much more efficiently (i.e., by smaller FSMs) by the use of guard functions. Moreover, guard functions are side-effect free by definition, which allows them to be evaluated in parallel in principle. This parallelism is lost by the FSM corresponding to the restricted model, where the guard functions are evaluated by action functions.

Note that if transitions of the basic model would be restricted to a single possible target mode, the transformation procedure described above would be no longer applicable. In this case, the basic model with guard functions would be more expressive than the restricted model without guard functions.

3.4 Related Work and Limitations

One of the first modeling approaches integrating FSMs with dataflow models is **charts* (pronounced "star charts") [GLL99]. The *charts approach focuses on the nesting of hierarchical FSMs within a variety of concurrency models, like dataflow, synchronous/reactive (SR), and discrete-event (DE). For this purpose, *charts pursue a "black box" approach, i.e., on each level of hierarchy, a set of connected actors is described, while actors are treated as black boxes. As will be seen, this approach is comparable to the hierarchical model introduced in Section 5.1. However, the FSM is only used to control the nested actors, but cannot be used to describe the behavior of the nested dataflow actors. To this end, a separate modeling formalism must be used. In order to embed DDF actors, the modeling formalism must describe these actors by means of firing rules. As the proposed approach is based on firing rules (cf. Definition 3.5), it could be used by hierarchical FSMs according to the *charts approach.

The CAL actor language presented in [EJ03] is part of the Ptolemy II project [EJL+03]. In CAL, actors are described by means of *actions*, which are similar to transitions of the proposed dataflow model. In particular, an action consists of an *input pattern* which specifies the tokens to be consumed from each input port, an *output pattern* which specifies the tokens to be produced on each output port, a *guard expression* which may evaluate token values and actor variables, and an *action body* which can modify the actor variables. Moreover, actions

may be scheduled by means of an FSM or regular expressions (which, of course, can be transformed into each other). The main difference between CAL and the proposed approach is that in CAL, the number of tokens consumed and produced inferred from the input patterns and output patterns may depend on actor variables. Thus, in the general case, it is not statically known how many tokens will be consumed and produced by an action. In fact, the number of tokens produced is known only after the action block has been executed, as the output pattern may depend on actor variables which are updated by the action block. In this case, the action block may stall if not enough space is available on some output ports. While CAL actions are therefore more expressive than the proposed approach, it also hinders analysis and hierarchical composition of actors. In contrast, the proposed dataflow model restricts transitions to a static communication behavior.

The Extended Codesign Finite State Machine (ECFSM) model has been presented in [SSL00]. The original Codesign Finite State Machine (CFSM) model used in POLIS [BGJ+97] is a special case of the ECFSM model. Both models, however, are refinements of the Abstract Codesign Finite State Machine (ACFSM) model, also presented in [SSL00]. An actor described by an ACFSM consists of a set of transitions, each of which specifies an *input enabling rate*, an *input consumption rate*, an *output production rate*, a *guard expression*, and a *set of vectors of expressions* which determine the values of produced tokens. While actor variables are not supported by ACFSMs, these can be represented by additional self-loop FIFO channels which contain the actor variables. While similar to the proposed dataflow model, an important difference is that transitions may "flush" an input channel by consuming all tokens from it. This is realized by means of a special input consumption rate I_n^{all}. While this feature can be used to model exception handling and reactions to disruptive events (e.g., errors and re-initializations), it leads to the same problems as described for CAL w.r.t. analysis and hierarchical composition of actors. In contrast, the proposed dataflow model restricts transitions to a static communication behavior. Transitions in ACFSMs are not governed by means of an FSM (despite the name). Instead, the selection of transitions is solely based on the input enabling rate and output production rate, as well as the result of the guard expression. This makes analysis of the communication behavior of an ACFSM more difficult, as all transitions of the ACFSM must be considered to be active in the general case.

FunState [TSZ+99; STG+01] uses nested components which are controlled by FSMs. Components consist of storage units, functions, or other components. Storage units consist of FIFO queues (of unbounded size) and registers. Functions and nested components are controlled by state machines. Transitions in FunState consist of a *predicate* and an *action*. The predicate specifies the required number of tokens in a queue, and can also evaluate token values or register values. While the guard predicates are similar to the proposed approach, actions use *events*

to activate components, i.e., functions or nested components. This may result in a non-sequential behavior of components, where a transition may be started before the previous one is finished. In turn, actor variables must be stored in a self-loop channel, and tokens must be consumed and produced atomically at the beginning and end of an action, respectively. In contrast, transitions of an actor have sequential semantics in the proposed dataflow model, and thus, stateful actors are permitted, and tokens can be consumed and produced at any time during the execution of a transition. Moreover, transitions of the proposed dataflow model have a static communication behavior, which may not be the case in FunState.

In [PSK+08], the enable-invoke dataflow (EIDF) model is introduced. Here, an actor also consists of *modes*. However, each mode is implicitly associated with only a single transition. Thus, in order to provide for DDF actors, an actor may have multiple active modes. The *enabling function* of an actor determines whether a mode is enabled or not. To this end, it is provided an active mode, and the number of tokens available on each input port. Note that it is mentioned that "each mode, when executed, consumes and produces a fixed number of tokens". This corresponds to the static communication behavior of transitions in the proposed dataflow model. In contrast to transitions of the proposed dataflow model (cf. 3.1), decisions depending on token values or actor variables are not supported. While possibly cumbersome as outlined in Section 3.3, this makes the model not less expressive if action functions can select the target mode. Indeed, this is supported by EIDF: the *invoking function* of an actor is provided an active mode (for which the enabling function returned $\top$), consumes and produces tokens, and determines a set of modes which should be active after the invoking function is finished. While actor variables are not supported by EIDF, these can be represented by additional self-loop FIFO channels which contain the actor variables. A less expressive model, namely core functional dataflow (CFDF), restricts the invoking function to return only a single mode instead of a set of modes. As each actor mode is associated with only a single transition, the deterministic CFDF is less expressive than EIDF, which can be used to model nondeterministic behavior.

Bluespec SystemVerilog (BSV) [RA04; ANRD04] is based on a synthesizable subset of SystemVerilog [Acc04]. In BSV, modules consist of a set of variables and a set of *rules*, which are based on guarded atomic actions. Modules expose interface methods which can be called by other modules. A rule is enabled if the guard predicate is true. When an enabled rule is selected for execution, its action is executed atomically. Rules may be executed in parallel if the result matches the result of a sequential execution of the rules. To this end, the BSV compiler analyzes the rules in a design and synthesizes hardware scheduling logic which controls the parallel execution of rules. While BSV is also based on guarded actions comparable to the proposed approach, the level of abstraction is lower

than the proposed approach. In particular, rules are already at register-transfer level (RTL) and are thus assumed to take only one clock cycle. In other words, rules describe the combinational logic between registers. In contrast, action functions (and guard functions) in the proposed dataflow model are expected to be synthesized to RTL by a high-level synthesis (HLS) tool, and can therefore require multiple clock cycles for execution. Note that the parallel execution of transitions in the proposed dataflow model is also allowed if the resulting model state corresponds to a sequential execution of the transitions in question as described in Section 3.1.1.

In StreamIt [TKA02], actors (called *filters*) consist of only a single action function which is executed when enough tokens and free places are available. Analogously to transitions in the proposed dataflow model, filters are constrained to static token consumption and production rates in order to improve analyzability. As a filter consists only of a single action function, guard functions are not necessary. In contrast, the proposed dataflow model provides for multiple transitions of an actor with possibly different (static) token consumption and production rates. While StreamIt allows some out-of-stream communication between filters and an occasional structural modification of the filter graph, filter graphs are restricted to (basic) filters with one input port and one output port, `Split` filters with one input port and two output ports, and `Join` filters with two input ports and one output ports. Thus, in contrast to the proposed approach, only limited graph topologies can be realized. Moreover, the proposed approach provides for actors with an arbitrary number of input ports and output ports.

In SysteMoC [FKH+08; FZK+11; FZHT11; FZHT13] actors are also described by means of an FSM similar to the proposed approach. In this case, the key difference can be found in the well-defined hierarchical compositionality of the proposed dataflow model as described in Section 5.1. In particular, hierarchical actors in the proposed dataflow model can be used to implement custom scheduling schemes, whereas hierarchical actors in SysteMoC are limited to a structural refinement, and the same dynamic scheduling scheme is used for all hierarchical actors. In SysteMoC, transitions may specify *enabling rates* separately from the token consumption and production rates. In other words, more tokens or free places may be required by a transition in order to be enabled than tokens are consumed and produced by the associated action function. In order to increase analyzability, transitions of the proposed dataflow model have no separate enabling rates, i.e., a transition is enabled if enough tokens and free places according to the values of cons and prod are available, and the associated action function must consume and produce exactly as many tokens. Note that $\text{peek}(p, t.f_g) \leq \text{cons}(p, t.f_a)$ for a given well-formed transition t and input port $p \in I$. These token consumption/production semantics adhere to the established token consumption/production semantics of less expressive dataflow models, like

SDF or cyclo-static dataflow (CSDF). An available implementation of SysteMoC is based on SystemC, while the current reference implementation of the proposed dataflow model is not based on SystemC. To this end, SysteMoC extends SystemC by guarded actions (cf. Definition 3.5) and custom FIFO channels in order to support the communication semantics of guard functions and action functions (cf. Definitions 3.3 and 3.4). However, the designer is supposed to follow a certain coding style which basically forbids the use of processes and channels except the custom FIFO channels in order to obtain an analyzable dataflow model. Thus, while the SystemC utility classes like logic data types, logic vectors, etc. can be used by models, SystemC is mainly used as a simulation vehicle. As the reference implementation of the proposed dataflow model separates the model from the simulation, arbitrary simulation back ends can be developed in principle, including a similar SystemC-based simulation back end.

The proposed dataflow model is based on *interleaved guarded actions*, i.e., only a single enabled transition of a given actor is chosen for execution at any one time. In contrast, *synchronous programming languages* like Quartz [Sch09], SIGNAL [LBBG86], LUSTRE [CPHP87], or Esterel [BG92] can be translated into *synchronous guarded actions* [BSS10], where *all* guarded actions of an actor are executed synchronously within each *macro step* of the system. In order to leverage the analysis, verification and synthesis tools available for models based on interleaved guarded actions, it is desirable to translate synchronous guarded actions into interleaved guarded actions as shown in [BBS11; GS13b; GS13a; KBS14]. In particular, the latter approach translates synchronous guarded actions into the SysteMoC model which has been reviewed above. Thus, the proposed dataflow model could also be used to represent these translated synchronous guarded actions.

4

Analysis

The dataflow model proposed in Chapter 3 is expressive enough to model DDF (i.e., nondeterministic) applications. As a consequence, design methods applicable to less expressive dataflow models, like the static scheduling of actors, cannot be used in the presence of such a highly expressive model. In order to use this potential, actors which adhere to less expressive dataflow models must be identified. In this chapter, some well-known dataflow MoCs are discussed. In particular, for each dataflow MoC, it is described how it can be represented by the proposed dataflow model, and how it can be extracted from a given actor implemented by means of the proposed dataflow model. Section 4.10 discusses related work and the limitations of the proposed classification approach. Concerning the exemplary design flow supported by the proposed dataflow model as shown in Figure 4.1, the proposed classification approach therefore supports the (automatic) decision-making process at system level.

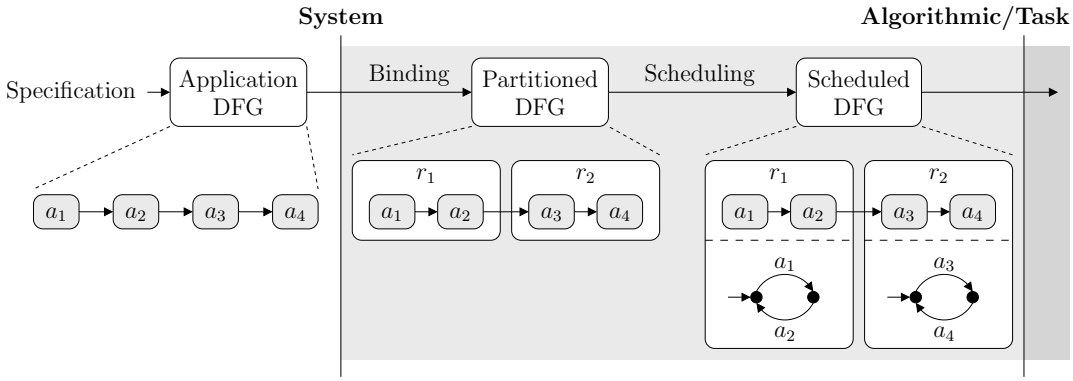


Figure 4.1: Design flow supported by the proposed model. Only the steps pertaining to system synthesis are shown (cf. Figure 2.2 on page 9).

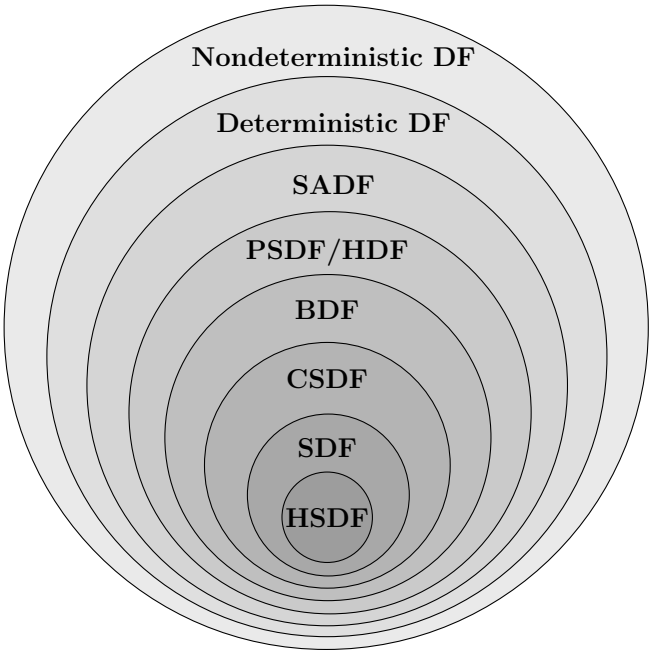


Figure 4.2: Dataflow models sorted by expressiveness.

4.1 Dataflow Models of Computation

Over the last decades, many dataflow MoCs have been developed. They are usually classified according to their *expressiveness*, i.e., which kind of applications can be modeled by using a given dataflow MoC. It can be observed that *analyzability* of dataflow MoCs is inversely related to their expressiveness, i.e., there are problems which are decidable for less expressive dataflow MoCs, but are not decidable for more expressive dataflow MoCs. As analyzability directly influences the Quality of Results (QoR) of the final product[4], dataflow MoCs with a high analyzability are usually desirable. For example, scheduling at compile time (*static scheduling*) is usually preferred over scheduling at run time (*dynamic scheduling*) in order to reduce the overhead incurred by the scheduling strategy. However, static schedules can only be computed for dataflow MoCs with limited expressiveness. Figure 4.2 shows some well-known dataflow MoCs, where larger circles denote more expressive MoCs [SGTB11]. These are discussed in the following sections.

[4]The Quality of Results (QoR) refers to the set of quality indicators of the final product like cost, performance, or power consumption.

4.2 (Homogeneous) Synchronous Dataflow

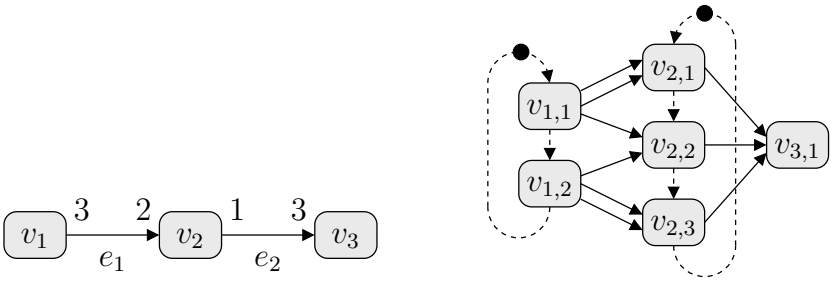
The synchronous dataflow (SDF) model has been introduced in [LM87]. SDF actors are characterized by the fact that they have *static* (i.e., data-independent) token consumption and production rates. In other words, an SDF actor, when fired, consumes a fixed number of tokens from each input port, and produces a fixed number of tokens on each output port. To avoid confusion, "synchronous" in this context refers to the static token consumption and production rates, and is therefore used in a different sense compared to *synchronous programming languages* like Quartz [Sch09], SIGNAL [LBBG86], LUSTRE [CPHP87], or Esterel [BG92]. In contrast, *asynchronous* actors exhibit data-dependent token consumption and production rates. Considering Figure 4.2, this is the case for the Boolean dataflow (BDF) model and the dataflow models of higher expressiveness.

An SDF graph $G = (V, E, \text{cons}, \text{prod}, \text{D})$ consists of a set of actors V, a set of unbounded channels $E \subseteq V \times V$, token consumption rates $\text{cons}\colon E \to \mathbb{N}$, token production rates $\text{prod}\colon E \to \mathbb{N}$, and a delay function $\text{D}\colon E \to \mathbb{N}_0$ which specifies the number of initial tokens on each channel. For *multi-rate* SDF graphs, the values of cons and prod are unconstrained. For *single-rate* SDF graphs, the values of cons and prod are constrained to be equal w.r.t. a given edge, i.e., $\forall e \in E : \text{cons}(e) = \text{prod}(e)$. For homogeneous synchronous dataflow (HSDF) graphs, the values of cons and prod are constrained to be 1, i.e., $\forall e \in E : \text{cons}(e) = 1 \wedge \text{prod}(e) = 1$.

Both SDF and HSDF graphs are special cases of *Petri nets* (in the sense of place/transition (P/T) nets) [Mur89]. In particular, HSDF graphs correspond to marked graphs (MGs) [CHEP71], while SDF graphs correspond to weighted marked graphs (WMGs) [TCCS92]. As we are more interested in the operational semantics of SDF and HSDF graphs, we refrain from a formal definition of Petri nets. Suffice to say, the key difference between SDF graphs and Petri nets is that SDF graphs are *conflict-free*, which means that once an actor is enabled (i.e., enough tokens are available on input ports), it cannot be disabled by the firing of a different actor. This is due to the fact that channels in SDF graphs have point-to-point semantics, i.e., they have exactly one source actor and one sink actor. In contrast, *places* in Petri nets may have more than one sink *transition*. Note that all dataflow MoCs examined in this chapter are conflict-free due to the point-to-point semantics of channels.

For SDF graphs, an important concept is the *repetition vector* $\boldsymbol{\gamma} \in \mathbb{N}^{|V|}$. The repetition vector $\boldsymbol{\gamma}$ denotes the smallest non-trivial vector of actor firings which return the SDF graph into its initial *state*. The state of an SDF graph is defined by the number of tokens on each channel. More formally, the repetition vector (if it exists) is the non-trivial solution to the *balance equations* which can be stated as follows:

$$\forall e = (v_i, v_j) \in E : \gamma_i \cdot \text{prod}(e) = \gamma_j \cdot \text{cons}(e) \tag{4.1}$$



a) SDF graph a) Equivalent HSDF graph

Figure 4.3: a) SDF graph with repetition vector $\boldsymbol{\gamma} = (2, 3, 1)$, and b) equivalent HSDF graph with duplicated actors according to $\boldsymbol{\gamma}$.

The balance equations can be solved efficiently in linear time complexity $O(|V| + |E|)$ by means of a depth-first search (DFS) of the SDF graph. If such a vector can be computed for a given SDF graph, the SDF graph is said to be *consistent*. Note that for single-rate SDF graphs and HSDF graphs, $\boldsymbol{\gamma} = \mathbf{1}$. For consistent SDF graphs, it is possible to determine a periodic static order schedule (PSOS) for the actor firings specified by $\boldsymbol{\gamma}$ if enough initial tokens have been placed on channels which comprise the cycles of the SDF graph. A PSOS is a total ordering of the actor firings after which the SDF graph returns into its initial state.

Example 4.1. Consider the SDF graph shown in Figure 4.3a. It consists of three actors v_1, v_2 and v_3, and two edges $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_3)$. When v_1 is fired, it produces three tokens on e_1. When v_2 is fired, it consumes two tokens from e_1, and produces one token on e_2. When v_3 is fired, it consumes three tokens from e_2. Thus, $\text{prod}(e_1) = \text{cons}(e_2) = 3$, $\text{cons}(e_1) = 2$, and $\text{prod}(e_2) = 1$. Note that $\text{D}(e_1) = \text{D}(e_2) = 0$, i.e., no initial tokens are placed on any channel. Obviously, the repetition vector is $\boldsymbol{\gamma} = (2, 3, 1)$, i.e., in order to return the SDF graph into its initial state, v_1 must be executed twice, v_2 three times, and v_3 only once. A possible PSOS in this case is $S_1 = \langle v_1^2, v_2^3, v_3 \rangle$, which fires v_1 twice, v_2 three times, and v_3 once. Such a PSOS is also called a single-appearance schedule (SAS), because each actor appears only once in S_1. Note that S_1 requires a maximal buffer size of 6, which can be verified by simulating the PSOS. Another possible PSOS is $S_2 = \langle \langle v_1, v_2 \rangle^2, v_2, v_3 \rangle = \langle v_1, v_2, v_1, v_2, v_2, v_3 \rangle$, which is not a SAS. However, in contrast to S_1, S_2 only requires a maximal buffer size of 4.

Every consistent SDF graph can be transformed into a functionally equivalent HSDF graph if we allow multiple edges between actors [NS99]. Basically, an actor v_i is duplicated γ_i times, and edges are added according to the token consumption and production rates of v_i. Additional edges are added

in order to serialize the firings of the duplicated instances of v_i. Note that this is only necessary if v_i has some internal actor state. In this case, the token passed between the duplicated instances of v_i represents the internal actor state. However, it can be shown that the number of duplicated actors of the resulting HSDF graphs grows exponentially with the number of actors $|V|$ of the SDF graph in the worst case: Consider an SDF graph with actors $V = \{v_1, v_2, \ldots, v_n\}$, edges $E = \{(v_1, v_2), \ldots, (v_i, v_{i+1}), \ldots, (v_{n-1}, v_n)\}$, token production rates $\text{prod}(e) = 2$, and token consumption rates $\text{cons}(e) = 1$. Then, it follows that $\boldsymbol{\gamma} = (1, 2^1, 2^2, \ldots, 2^{n-1})$. Therefore, the number of duplicated actor instances is $\sum_{i=0}^{n-1} 2^i = 2^n - 1 = 2^{|V|} - 1$. Thus, algorithms which transform an SDF into an HSDF graph for analytic purposes are typically only feasible for small SDF graphs.

Example 4.2. Consider again the SDF graph shown in Figure 4.3a. In order to obtain an equivalent HSDF graph, the actors v_1, v_2, and v_3 have been duplicated according to $\boldsymbol{\gamma} = (2, 3, 1)$ (cf. Figure 4.3b). Note that the dashed edges are only required if v_1 and v_2 have some internal state. In this case, the initial tokens placed on the incoming edges of $v_{1,1}$ and $v_{2,1}$ represent the initial internal state.

As both HSDF and SDF are special cases of the CSDF model, the discussion of the model representation and analysis is postponed to the next section.

4.3 Cyclo-Static Dataflow

The cyclo-static dataflow (CSDF) model has been introduced in [BELP96]. A CSDF actor consists of one or more *phases* as specified by the function $\text{P}\colon V \to \mathbb{N}$. In each phase, a CSDF actor may have different (but static) token consumption and production rates. Thus, in contrast to SDF actors, the token consumption rates are defined as $\text{cons}\colon E \times \mathbb{N} \to \mathbb{N}_0$, and the token production rates are defined as $\text{prod}\colon E \times \mathbb{N} \to \mathbb{N}_0$. Note that for any edge $e = (v_i, v_j) \in E$, $\text{prod}(e, n) = \text{prod}(e, n + k \cdot \text{P}(v_i))$, and $\text{cons}(e, n) = \text{cons}(e, n + k \cdot \text{P}(v_j))$, with $n \in \mathbb{N}$ and $k \in \mathbb{N}_0$. Each actor v is initially in phase 1. After firing v, it is in phase 2, etc. When v is in phase $\text{P}(v)$, firing v returns it to phase 1. If a given CSDF actor v consists only of a single phase, i.e., $\text{P}(v) = 1$, v is actually an SDF actor. Remember that HSDF actors are additionally constrained to token consumption and production rates of 1.

In order to determine the repetition vector $\boldsymbol{\gamma}$ of a CSDF graph, the following modified balance equations must be solved first, resulting in a vector $\boldsymbol{\gamma}'$:

$$\forall e = (v_i, v_j) \in E : \gamma'_i \sum_{n=1}^{P(v_i)} \text{prod}(e, n) = \gamma'_j \sum_{n=1}^{P(v_j)} \text{cons}(e, n) \tag{4.2}$$

Note that $\boldsymbol{\gamma}'$ corresponds to the number of complete cycles for each actor. As each cycle of an actor v_i consists of $P(v_i)$ firings, it follows that $\gamma_i = \gamma'_i \cdot P(v_i)$. Thus, $\boldsymbol{\gamma} = (\gamma'_1 \cdot P(v_1), \ldots, \gamma'_n \cdot P(v_n))$.

Example 4.3. Consider the CSDF graph in Figure 4.4. Structurally, it is equivalent to the SDF graph shown in Figure 4.3a. However, actors v_1 and v_3 now consist of three phases (i.e., $\mathrm{P}(v_1) = \mathrm{P}(v_3) = 3$), while actor v_2 consists of two phases (i.e., $\mathrm{P}(v_2) = 2$). Note that the values of cons and prod are annotated as tuples to the edges. For example, v_2 produces one token on e_2 in its first phase, but produces no tokens on e_2 in its second phase. As the sums specified in Equation (4.2) correspond to the token consumption and production rates of the SDF graph from Figure 4.3a, it follows that $\boldsymbol{\gamma}' = (2, 3, 1)$ (cf. Example 4.1). This results in a repetition vector $\boldsymbol{\gamma} = (2 \cdot 3, 3 \cdot 2, 1 \cdot 3) = (6, 6, 3)$ of the CSDF actor. A possible PSOS is $S = \langle\langle v_1, v_2\rangle^2, v_3\rangle^3$, which requires only a maximal buffer size of 1, and additionally, is a SAS.

4.3.1 Representation

Given a CSDF graph $G = (V, E, \text{cons}, \text{prod}, \mathrm{D}, \mathrm{P})$ which consists of a set of actors V, a set of unbounded channels $E \subseteq V \times V$, token consumption rates $\text{cons}\colon E \times \mathbb{N} \to \mathbb{N}_0$, token production rates $\text{prod}\colon E \times \mathbb{N} \to \mathbb{N}_0$, a delay function $\mathrm{D}\colon E \to \mathbb{N}_0$ which specifies the number of initial tokens on each channel, and a function $\mathrm{P}\colon V \to \mathbb{N}$ which specifies for each actor the number of phases.

Then, for a given actor $v \in V$, a functionally equivalent actor a in the modeling approach presented in Section 3.1 can be constructed as follows: First, the input ports I and output ports O of a correspond to the incoming and outgoing edges of v, respectively. For each phase $k \in \mathbb{N}, 1 \leq k \leq \mathrm{P}(v)$, a mode $m_k \in M$ is allocated, with the initial mode being m_1. For each mode m_k, a single outgoing transition t_k is allocated as follows: The source mode of t_k is m_k, and the (only possible) target mode of t_k is m_{k+1} if $k < \mathrm{P}(v)$, or m_1 if $k = \mathrm{P}(v)$. Thus, modes and transitions form a cycle in the FSM (cf. Figure 4.5). The transitions do not

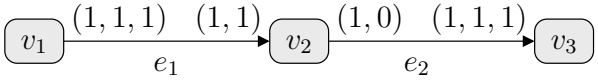
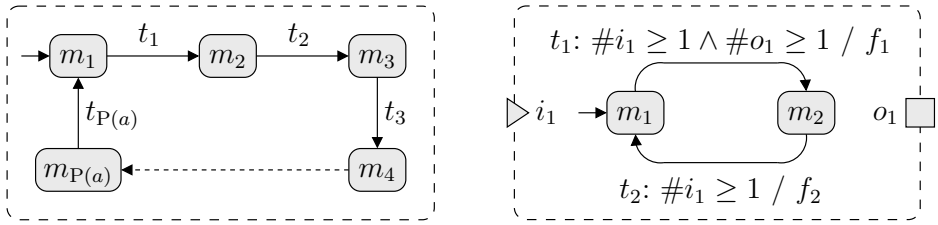


Figure 4.4: CSDF graph



a) Normalized FSM of a CSDF actor b) Normalized example FSM

Figure 4.5: The normalized representation of the FSM of a CSDF actor a with $\mathrm{P}(a)$ phases is shown in (a). Note that each mode m_i has exactly one outgoing transition t_i with $t_i.f_g = f_\top$. The normalized example FSM of the CSDF actor v_2 from Figure 4.4 with two phases is shown in (b).

have a guard function, i.e., $\forall t \in T : t.f_g = f_\top$. Concerning the action function, each transition is associated with the corresponding firing of the CSDF actor, i.e., $\forall t_k \in T : t_k.f_a = f_k$, where f_k denotes the functionality associated with the firing of the CSDF actor in phase k. For the token consumption rates of an action function f_k, it follows that for each input port $p \in I$ corresponding to an incoming edge $e \in E$, $a.\mathrm{cons}(p, f_k) = G.\mathrm{cons}(e, k)$. Analogously, for the token production rates of an action function f_k, it follows that for each output port $p \in O$ corresponding to an outgoing edge $e \in E$, $a.\mathrm{prod}(p, f_k) = G.\mathrm{prod}(e, k)$.

Example 4.4. For the CSDF actor v_2 from Figure 4.4, the resulting normalized actor FSM is shown in Figure 4.5b. Corresponding to the first phase, transition t_1 consumes a token from i_1, and produces a token on o_1. Corresponding to the second phase, transition t_2 only consumes a token from i_1.

The channels of the proposed dataflow model are bounded. In this respect, the CSDF model is more expressive than the proposed dataflow model. However, an unbounded accumulation of tokens on a channel is typically considered undesired behavior in real-world applications. Thus, we can assume that only consistent CSDF graphs are considered, for which a repetition vector can be calculated. In this case, an unbounded accumulation of tokens on a channel is not possible, and any schedule (if one exists) can be implemented by means of bounded channels.

4.3.2 Identification

While it is possible to transform any CSDF actor (but not any CSDF graph) into the proposed dataflow model as described in the previous section, it is much more difficult to check if a given actor shows CSDF behavior. This is due to the fact that, in the general case, actor FSMs are not normalized as shown in

Figure 4.5. Given an arbitrary actor FSM in the proposed dataflow model, the question therefore is: does every possible transition path from the initial mode m_0 exhibit the same (cyclo-)static communication behavior? The basic idea of the classification algorithm presented in [ZFHT08] can be summarized as follows:

- During the first phase, the *smallest repeatable communication pattern* is identified. This is done by successively *constructing* and *validating* communication patterns. If every possible transition path from the initial mode m_0 adheres to such a constructed communication pattern, a repeatable communication pattern has been found, and the first phase is finished successfully. If no more communication patterns can be constructed, the actor does not show CSDF behavior, and the remaining phases are skipped.
- During the second phase, the repeatable communication pattern is *partitioned* into phases. If the chosen partitioning is successfully validated against every possibly transition path from the initial mode m_0, the actor shows CSDF behavior. Otherwise, the actor does not show CSDF behavior, and the last phase is skipped.
- During the third phase, the actor FSM is transformed into the normalized representation as shown in Figure 4.5a according to the partitioning of the repeatable communication pattern into phases as determined during the previous phase.

Before illustrating the proposed classification algorithm by means of a small example, we first introduce some notations which are used in the following: A static communication pattern is specified by the vectors **cons** and **prod**. While $\mathbf{cons} = (c_1, \ldots, c_{|I|})$ specifies that c_k tokens are consumed from the input port $i_k \in I$, $\mathbf{prod} = (p_1, \ldots, p_{|O|})$ specifies that p_k tokens are produced on the output port $o_k \in O$. In order to simplify the notation, **cons** and **prod** are also written as a combined vector $\mathbf{cp} = \mathbf{cons} \frown \mathbf{prod}$. In the following, the vector-based token consumption and production rates are also used for transitions. To this end, the token consumption rates of a transition t are referred to as $\mathbf{cons}(t) = (\mathrm{cons}(i_1, t.f_a), \ldots, \mathrm{cons}(i_{|I|}, t.f_a))$, while the token production rates of t are referred to as $\mathbf{prod}(t) = (\mathrm{prod}(o_1, t.f_a), \ldots, \mathrm{prod}(o_{|O|}, t.f_a))$. Analogously to $\mathbf{cp} = \mathbf{cons} \frown \mathbf{prod}$, we use $\mathbf{cp}(t) = \mathbf{cons}(t) \frown \mathbf{prod}(t)$. Note that comparison operations between two vectors $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$ are defined as follows: $\mathbf{a} \geq \mathbf{b} \Leftrightarrow a_i \geq b_i, 1 \leq i \leq n$, and $\mathbf{a} > \mathbf{b} \Leftrightarrow \mathbf{a} \geq \mathbf{b} \wedge \mathbf{a} \neq \mathbf{b}$.

A sequence of transitions $p = \langle t_1, \ldots, t_n \rangle$ is a *transition path* if the set of possible target modes of a transition t_i contains the source mode of transition t_{i+1}, i.e., $\forall i, 1 \leq i < n : t_i.M' \ni t_{i+1}.m$. A transition path $p = \langle t_1, \ldots, t_n \rangle$ is a *transition cycle* if the set of possible target modes of transition t_n contains the source mode of transition t_1, i.e., $t_n.M' \ni t_1.m$. A transition path p is

an *elementary transition path* if p does not contain any cyclic subpaths. A transition cycle p is an *elementary transition cycle* if p does not contain any cyclic subpaths (except p itself). In the following, the vector-based token consumption and production rates are also used for transition paths $p = \langle t_1, \ldots, t_n \rangle$, i.e., $\mathbf{cons}(p) = \sum_{i=1}^{n} \mathbf{cons}(t_i)$, and $\mathbf{prod}(p) = \sum_{i=1}^{n} \mathbf{prod}(t_i)$. Analogously to $\mathbf{cp} = \mathbf{cons} \frown \mathbf{prod}$, we use $\mathbf{cp}(p) = \mathbf{cons}(p) \frown \mathbf{prod}(p)$.

Example 4.5. Consider the normalized actor FSM in Figure 4.5b. Assume that a communication pattern $\mathbf{cp} = (2, 1)$ has been constructed (or guessed). In this case, two tokens are consumed from input port i_1, and one token is produced on output port o_1. Starting from the initial mode (m_1 in this case), $\mathbf{cp}$ is validated against all possible transition paths. It should be obvious that the only possible transition path is $p = \langle t_1, t_2 \rangle$, and that $\mathbf{cp}(p) = \mathbf{cp}(t_1) + \mathbf{cp}(t_2) = (1, 1) + (1, 0) = (2, 1) = \mathbf{cp}$. Thus, a repeatable communication pattern $\mathbf{cp} = (2, 1)$ has been found. Note that $\mathbf{cp}$ spans multiple transitions. During the next phase of the classification algorithm, $\mathbf{cp}$ is partitioned into phases. This is done by means of a transition path p from the initial mode m_1 with $\mathbf{cp}(p) = \mathbf{cp}$. Obviously, $p = \langle t_1, t_2 \rangle$ in this case. Note that the proposed partitioning algorithm creates as few phases as possible, but as many as necessary in order to guarantee a deadlock-free execution of the transformed model, where the actor in question is treated as a CSDF actor. In this case, the resulting partitioning of p is $p = p_1 \frown p_2$, where $p_1 = \langle t_1 \rangle$, and $p_2 = \langle t_2 \rangle$. Obviously, the chosen partitioning is successfully validated against all possible transition paths from m_1. Thus, we conclude that the actor corresponds to a CSDF actor with two phases. Finally, as the actor FSM is already normalized, the third phase of the classification algorithm outputs the very same actor FSM.

The example shows that for normalized actor FSMs, the actor classification is trivially accomplished. In the following, however, arbitrary actor FSMs are analyzed. To this end, we first describe some requirements which the given actor must satisfy such that the introduction of deadlocks into the transformed model (which treats a given actor as a CSDF actor) is avoided.

Requirement 4.1 (Liveness)**.** Let $T(m) = \{t \in T \mid t.m = m\}$ be the set of outgoing transitions of a mode $m \in M$. We require that all modes $m \in M$ have some outgoing transitions (i.e., $T(m) \neq \emptyset$), and that at least one of these transitions is eventually enabled if $m_{\text{cur}} = m$. Note that for a mode $m \in M$ where $T(m) = \emptyset$, the classification procedure assumes that the FSM is dead, and immediately discards the given actor as not being a CSDF actor, as CSDF actors are expected to fire an infinite number of times. However, for a mode $m \in M$ where $T(m) \neq \emptyset$, this problem is undecidable in the general case due to the presence of guard functions. Thus, the user has to ensure that this requirement is met.

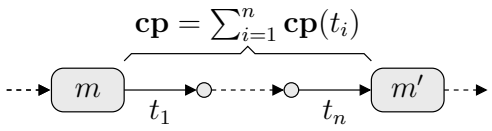


Figure 4.6: The static communication pattern may span multiple transitions.

Requirement 4.2 (Active input/output behavior)**.** An actor may have transitions which do neither consume nor produce any tokens. While this is not a problem in the general case, we require that, eventually, the actor must be blocked on missing tokens or free places. This requirement ensures that the merging of transitions in order to determine the normalized FSM does not introduce infinite loops into the actions of the merged transitions. Note that this is basically the same constraint imposed by SystemC on processes, which must relinquish control to the simulation kernel eventually. Thus, actors derived from well-formed SystemC modules as described in Section 2.3 should meet this requirement.

Smallest Repeatable Communication Pattern

The first goal of the classification algorithm is to verify whether or not all possible transition paths from the initial mode adhere to a given static communication behavior **cp**, which is formally defined as follows:

Definition 4.1. A transition path p adheres to a given static communication behavior **cp** if the sum of the token consumption and production rates of the transitions of p are equal to the token consumption and production rates as specified by **cp**, i.e., if $\mathbf{cp}(p) = \mathbf{cp}$.

In order to increase the number of actors which are identified as a CSDF actor, we allow p to span multiple transitions (cf. Figure 4.6). It should be noted that Condition 4.1 is checked incrementally while traversing the FSM.

The validation of a static communication pattern **cp** for a mode m is summarized by Algorithm 4.1. Here, $\mathbf{cp}'$ denotes the number of remaining tokens which must still be consumed and produced by subsequent transitions (remember that the communication pattern may span multiple transitions).

Procedure ValidateMode (cf. lines 1-14) validates all outgoing transitions of a mode m. If m starts a new iteration of the static communication pattern, i.e., if $\mathbf{cp} = \mathbf{cp}'$, we first check if m is already marked as validated (cf. lines 2-3). In this case, we assume that the validation of m is successful, and return $\top$ (cf. line 4). Otherwise, m is marked as validated (cf. line 6). Subsequently, the outgoing transitions of m are validated (cf. lines 8-12). Here, if ValidateTransition returns $\bot$, ValidateMode also returns $\bot$. If all outgoing transitions have been successfully validated, ValidateMode returns $\top$ (cf. line 13). In this case, m adheres to the static communication pattern **cp**.

Algorithm 4.1 Validation of the static communication pattern

```
 1: procedure ValidateMode(Mode m, cp, cp′)
 2:     if cp = cp′ then                          ▷ Mode m starts a new iteration
 3:         if m is marked as validated then
 4:             return ⊤
 5:         end if
 6:         Mark m as validated
 7:     end if
 8:     for all t ∈ T(m) do                       ▷ Process outgoing transitions of m
 9:         if ValidateTransition(t, cp, cp′) = ⊥ then
10:             return ⊥
11:         end if
12:     end for
13:     return ⊤
14: end procedure

15: procedure ValidateTransition(Transition t, cp, cp′)
16:     if cp(t) ⋠ cp′ then                                        ▷ Condition 4.1
17:         return ⊥
18:     end if
19:     cp′ ← cp′ − cp(t)
20:     if cp′ = 0 then                                  ▷ Transition path finished
21:         cp′ ← cp
22:     end if
23:     for all m′ ∈ t.M′ do                  ▷ Process possible target modes of t
24:         if ValidateMode(m′, cp, cp′) = ⊥ then
25:             return ⊥
26:         end if
27:     end for
28:     return ⊤
29: end procedure
```

Procedure ValidateTransition (cf. lines 15-29) validates a transition t as follows: First, Condition 4.1 is checked. Condition 4.1 is violated if t does not consume and produce less or equal tokens than specified by $\mathbf{cp}'$, i.e., if $\mathbf{cp}(t) \not\leq \mathbf{cp}'$ (cf. lines 16–18). If Condition 4.1 is violated, the FSM does not adhere to the static communication pattern $\mathbf{cp}$, and $\bot$ is returned. Otherwise, $\mathbf{cp}'$ is updated as follows: First, the tokens consumed and produced by t are subtracted from $\mathbf{cp}'$ (cf. line 19). If no tokens remain, i.e., if $\mathbf{cp}' = \mathbf{0}$, transition t successfully completed the validation of a transition path, and $\mathbf{cp}'$ is reset to $\mathbf{cp}$ (cf. lines 20–22). Subsequently, the possible target modes of t are processed analogously to

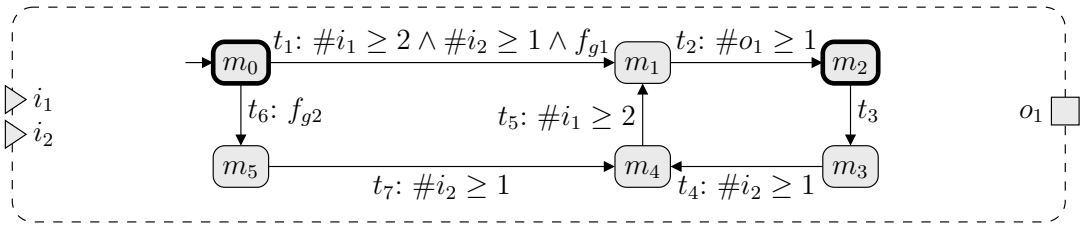
procedure ValidateMode (cf. lines 23–27). If all possible target modes have been successfully validated, ValidateTransition returns $\top$ (cf. line 28).

It should be noted that Algorithm 4.1 as given does not terminate if there are transition cycles p that do not consume or produce any tokens, i.e., if $\mathbf{cp}(p) = \mathbf{0}$. In this case, Condition 4.1 is never violated (cf. lines 16–18), and $\mathbf{cp}'$ is never decremented (cf. line 19), with the effect that p is traversed an infinite number of times. However, as an infinite traversal of p must not happen when executing the actor according to Requirement 4.2, such transition cycles can be safely ignored. Note that the detection of such transition cycles requires some additional bookkeeping, which has been omitted in Algorithm 4.1 for the sake of clarity.

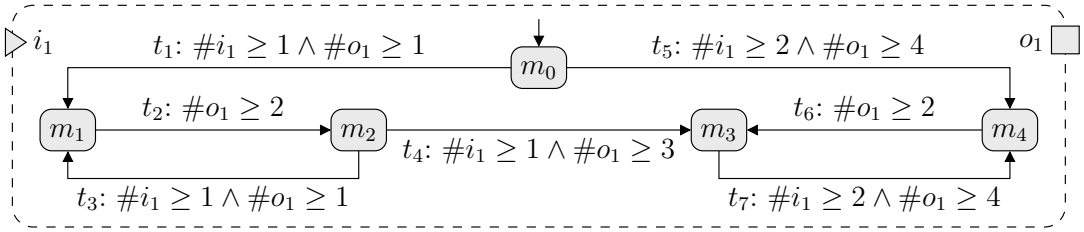
As each actor FSM starts in the initial mode m_0, the validation of the FSM is initiated by ValidateMode$(m_0, \mathbf{cp}, \mathbf{cp})$.

Example 4.6. Consider the actor FSM in Figure 4.7a. The corresponding actor has two input ports i_1 and i_2, and one output port o_1. First, assume that $\mathbf{cons} = (c_1, c_2) = (0, 1)$ and $\mathbf{prod} = (p_1) = (0)$, i.e., $\mathbf{cp} = (0, 1, 0)$. In this case, starting from m_0, validation fails for transition t_1, as $\mathbf{cp}(t_1) = (2, 1, 0) \not\leq (0, 1, 0) = \mathbf{cp}'$.

Observing that t_1 consumes two tokens from i_1, we now set $\mathbf{cp} = (2, 1, 0)$. Now, validation succeeds, e.g., for the transition path $\langle t_1 \rangle$. In this case, mode



a) Example FSM 1



b) Example FSM 2

Figure 4.7: Example FSMs with a static communication behavior (a), and a non-static communication behavior (b). For the sake of clarity, action functions have been omitted.

m_1 starts a new iteration of $\mathbf{cp}$. Obviously, validation now fails for transition t_2, as $\mathbf{cp}(t_2) = (0,0,1) \not\leq (2,1,0) = \mathbf{cp}'$.

Observing that t_2 produces a token on o_1, we now set $\mathbf{cp} = (2,1,1)$. Now, validation succeeds, e.g., for the transition path $\langle t_1, t_2 \rangle$. In this case, mode m_2 starts a new iteration of $\mathbf{cp}$. As the only transition path $\langle t_3, t_4, t_5, t_2 \rangle$ from m_2 also adheres to the static communication pattern $(2,1,1)$, m_2 is successfully validated. As the only remaining transition path from m_0, namely $\langle t_6, t_7, t_5, t_2 \rangle$ also adheres to the static communication pattern $(2,1,1)$, and m_2 is already marked as validated, validation of m_0 is completed successfully. Thus, the FSM has been successfully validated for $\mathbf{cp} = (2,1,1)$.

With regard to the second phase of the classification algorithm, assume that $\mathbf{cp}$ is partitioned into only one phase. In this case, we can conclude that the actor FSM shows SDF behavior, and that each firing consumes two tokens from i_1, one token from i_2, and produces one token on o_1.

Now, consider the actor FSM in Figure 4.7b. The corresponding actor has one input port i_1 and one output port o_1. In this case, the FSM cannot be successfully validated for any values $\mathbf{cp}$.

Until now, the construction of communication patterns has been neglected. As we are interested in the *smallest* repeatable communication pattern, the chosen construction algorithm can be outlined as follows: First, a transition cycle c is selected which is reachable from m_0 by means of a (possibly empty) transition path p, and consumes or produces some tokens, i.e., $\mathbf{cp}(c) \neq \mathbf{0}$. The resulting infinite transition path $p_\infty = p^\frown c^\frown c^\frown \ldots = \langle t_1, \ldots \rangle$ from the initial mode m_0 is then used to construct the communication patterns. To this end, the first transition t_i of p_∞ is determined which consumes or produces some tokens, i.e., $\mathbf{cp}(t_i) \neq \mathbf{0}$ and $\forall k, 0 < k < i : \mathbf{cp}(t_k) = \mathbf{0}$. Note that according to the construction of p_∞, such a transition must exist. For this first transition, we initialize $\mathbf{cp} \leftarrow \mathbf{cp}(t_i)$, and try to validate the FSM according to Algorithm 4.1. If validation succeeds, the actor FSM adheres to the static token consumption and production pattern $\mathbf{cp}$.

If validation fails, the basic idea is to *enlarge* the static communication pattern $\mathbf{cp}$. To this end, the next transition t_j, $j > i$ of p_∞ is determined which consumes or produces some tokens, i.e, $\mathbf{cp}(t_j) \neq \mathbf{0}$ and $\forall k, i < k < j : \mathbf{cp}(t_k) = \mathbf{0}$. Note that according to the construction of p_∞, such a transition must exist. Then, the token consumption and production rates of t_j are added to $\mathbf{cp}$, i.e, $\mathbf{cp} \leftarrow \mathbf{cp} + \mathbf{cp}(t_j)$. Now, the FSM is validated again according to Algorithm 4.1, and so on. This pattern construction scheme ensures that the smallest repeatable communication pattern is found (if it exists).

Theorem 4.1. At least one cyclic transition path c exists which is reachable from m_0 by means of a (possibly empty) transition path p, and consumes or produces some tokens.

Proof. As we only consider finite actor FSMs, the existence of a transition cycle c which is reachable from m_0 by means of a (possibly empty) transition path p directly follows from the finite number of actor modes and Requirement 4.1, which states that all modes must have at least one outgoing transition.

Now assume that no transition cycle c reachable from m_0 consumes or produces any tokens. Then, a necessary condition to satisfy Requirement 4.2 is that there must exist at least one transition t, such that t is reachable from m_0 and consumes or produces some tokens. Given such a transition t, Requirement 4.2 is satisfied only if $t.m$ is reachable from all transition cycles c. However, this means that t is part of a transition cycle reachable from m_0. This contradicts the assumption that no transition cycle c reachable from m_0 consumes or produces any tokens. Thus, at least one transition cycle c reachable from m_0 exists which consumes or produces some tokens. □

Example 4.7. Consider the FSM shown in Figure 4.7a. A transition path according to Theorem 4.1 is $p_1 = p_1' \frown c_1 \frown c_1 \frown \ldots$, where $p_1' = \langle t_6, t_7 \rangle$ and $c_1 = \langle t_5, t_2, t_3, t_4 \rangle$. In this case, the communication patterns constructed from p_1 are $\mathbf{cp}_1 = (0, 1, 0)$, $\mathbf{cp}_2 = (2, 1, 0)$, $\mathbf{cp}_3 = (2, 1, 1)$, and so on. Note that $\mathbf{cp}_3$ is successfully validated by Algorithm 4.1 (cf. Example 4.6).

Now, consider the FSM shown in Figure 4.7b. A transition path according to Theorem 4.1 is $p_2 = p_2' \frown c_2 \frown c_2 \frown \ldots$, where $p_2' = \langle t_1 \rangle$ and $c_2 = \langle t_2, t_3 \rangle$. In this case, the communication patterns constructed from p_2 are $\mathbf{cp}_1 = (1, 1)$, $\mathbf{cp}_2 = (1, 3)$, $\mathbf{cp}_3 = (2, 4)$, $\mathbf{cp}_4 = (2, 6)$, $\mathbf{cp}_5 = (3, 7)$, $\mathbf{cp}_6 = (3, 9)$, and so on. As the reader can easily verify, none of these communication patterns is successfully validated by Algorithm 4.1. In the following, it is shown that after $\mathbf{cp}_6$ has been processed, no more communication patterns need to be constructed.

As an infinite number of communication patterns can be constructed from the selected transition path p_∞ in principle, the classification algorithm never terminates if no static communication pattern exists for which the actor FSM can be successfully validated. In order to derive a termination criterion, we first define the notion of *cycle compatibility* and *path compatibility*:

Theorem 4.2 (Cycle compatibility)**.** Given two transition cycles c_1 and c_2 whose transitions do consume or produce some tokens, i.e., $\mathbf{cp}(c_1) \neq \mathbf{0}$ and $\mathbf{cp}(c_2) \neq \mathbf{0}$. If $\nexists r_1, r_2 \in \mathbb{N} : r_1 \, \mathbf{cp}(c_1) = r_2 \, \mathbf{cp}(c_2)$, then it is not possible to successfully validate the FSM. Note that in particular, the trivial solution $r_1 = 0$ and $r_2 = 0$ is not a feasible solution.

Note that transition cycles c whose transitions do not consume or produce any tokens are ignored as they cannot be iterated an infinite number of times due to Requirement 4.2.

Proof. Assume that for two transition cycles c_1 and c_2, a non-trivial solution cannot be found, but that the FSM is successfully validated for a static communication pattern $\mathbf{cp}$. Then, according to Condition 4.1, it follows that each cycle can be traversed one or more times in order to complete one or more iterations of the static communication pattern. Thus, for c_1 and c_2, it follows that

$$\exists a_1, a_2 \in \mathbb{N}\colon a_1\,\mathbf{cp}(c_1) = a_2\mathbf{cp}$$
$$\exists b_1, b_2 \in \mathbb{N}\colon b_1\,\mathbf{cp}(c_2) = b_2\mathbf{cp}$$

Thus, it follows that

$$\mathbf{cp} = \frac{a_1}{a_2}\,\mathbf{cp}(c_1) = \frac{b_1}{b_2}\,\mathbf{cp}(c_2)$$

We now set $r_1 = a_1 b_2$, and $r_2 = a_2 b_1$, and note that $r_1 > 0$ and $r_2 > 0$. Then, it follows that $r_1\,\mathbf{cp}(c_1) = r_2\,\mathbf{cp}(c_2)$. This is a contradiction to the assumption that a non-trivial solution cannot be found. □

Note that this property can be checked prior to validation by identifying the set of all elementary transition cycles C of the FSM, which can be efficiently done in $O((|M| + |T|)(|C| + 1))$ according to [Joh75]. Thus, FSMs which do not adhere to Theorem 4.2 are easily identified, and validation aborted.

Example 4.8. Consider the FSM shown in Figure 4.7a. The only elementary transition cycle is $c = \langle t_2, t_3, t_4, t_5\rangle$. Thus, Theorem 4.2 is trivially satisfied.

Now, consider the FSM shown in Figure 4.7b. The elementary transition cycles are $c_1 = \langle t_2, t_3\rangle$, and $c_2 = \langle t_6, t_7\rangle$. It follows that $\mathbf{cp}(c_1) = (1, 3)$, and $\mathbf{cp}(c_2) = (2, 6)$. It follows that $r_1 = 2$, and $r_2 = 1$. Thus, Theorem 4.2 is also satisfied. However, we know that the FSM does not show a static communication behavior.

The example shows that Theorem 4.2 is only a necessary condition for successful validation. In order for a static communication pattern to exist which is successfully validated, any transition path from m_0 by which a cycle is reached must also be considered. In order to define the notion of *path compatibility*, we assume the scenario shown in Figure 4.8, i.e., a mode m_1 of a transition cycle $c = \langle t_1, \ldots, t_n\rangle$ is reached by a transition path p' from m_0.

A static communication pattern $\mathbf{cp}$ is *repeatable* from a mode m in c if starting from m, a transition path p exists which contains only transitions of c such that

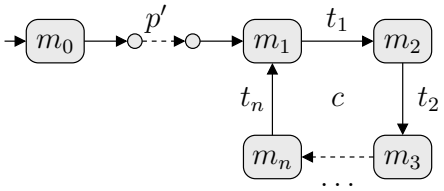


Figure 4.8: Path compatibility

$\mathbf{cp}(p) = \mathbf{cp}$, and if $\mathbf{cp}$ is repeatable from the mode m' of c which is reached by the last transition of p. This property is easily checked by traversing the transitions of c until either an incompatible transition is reached, or a mode is reached which has already been validated successfully. Note that this procedure is similar to Algorithm 4.1, but is constrained to a local FSM traversal of the cycle c. In the following, we use repeatable$(\mathbf{cp}, c, m) \in \{\top, \bot\}$ to denote whether a static communication pattern $\mathbf{cp}$ is repeatable in cycle c starting from a mode m of c. If repeatable$(\mathbf{cp}, c, m) = \top$, it follows that $\exists r_1, r_2 \in \mathbb{N} : r_1\mathbf{cp} = r_2\,\mathbf{cp}(c)$, i.e., in order to complete r_1 iterations of $\mathbf{cp}$ in c, c must be traversed r_2 times. For example, c may be traversed once to complete multiple iterations of $\mathbf{cp}$, or c may be traversed multiple times to complete a single iteration of $\mathbf{cp}$. Note that the existence of factors r_1 and r_2 is a necessary, but not a sufficient condition for the static communication pattern to be repeatable in c.

Theorem 4.3. Given a static communication pattern $\mathbf{cp}$, a cycle c, and mode m traversed by c. Then, repeatable$(\mathbf{cp}, c, m) = \top$ if (and only if) repeatable$(\mathbf{cp} + \mathbf{cp}(c), c, m) = \top$.

Proof. Theorem 4.3 follows directly from the fact that $\mathbf{cp}(c)$ is obviously a repeatable pattern in c from any mode m traversed by c. Nevertheless, we illustrate this property by means of Figure 4.9, which shows a repeatable pattern with $r_1 = 3$ and $r_2 = 4$, i.e., cycle c is traversed 4 times in order to complete 3 iterations of $\mathbf{cp} + \mathbf{cp}(c)$. In the following, we use $\mathbf{ecp} = \mathbf{cp} + \mathbf{cp}(c)$ to refer to the larger static communication pattern.

We first show that repeatable$(\mathbf{cp}, c, m) = \top$ if repeatable$(\mathbf{ecp}, c, m) = \top$. As $\mathbf{ecp} \geq \mathbf{cp}$, it follows that a transition subpath corresponding to one iteration of cycle c is contained in each of the r_1 iterations of $\mathbf{ecp}$. For example, for the first iteration of $\mathbf{ecp}$ in Figure 4.9 which corresponds to the transition path $p_1 = \langle t_1, t_2, t_3, t_4, t_1, t_2\rangle$, three such subpaths exists, namely $p_{1,1} = \langle t_1, t_2, t_3, t_4\rangle$, $p_{1,2} = \langle t_2, t_3, t_4, t_1\rangle$, and $p_{1,3} = \langle t_3, t_4, t_1, t_2\rangle$.

Removing an arbitrary transition subpath corresponding to $\mathbf{cp}$ from each iteration of $\mathbf{ecp}$, some transitions remain for each iteration of $\mathbf{ecp}$, and they form again a transition path. For example, removing $p_{1,2}$ from p_1, the residual transition path corresponds to $p'_1 = \langle t_1, t_2\rangle$. If instead, $p_{1,3}$ is removed from p_1,

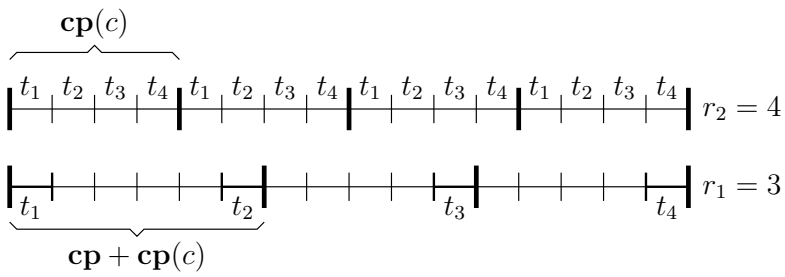


Figure 4.9: repeatable($\mathbf{cp}, c, m$) = $\top$ iff repeatable($\mathbf{cp} + \mathbf{cp}(c), c, m$) = $\top$.

the residual transition path is again $p'_1 = \langle t_1, t_2 \rangle$. In fact, as always a whole iteration of c is eliminated, these residual transition paths must always be the same. Note that if all residual transition paths p'_k of all r_1 iterations of **ecp** are concatenated, they correspond to multiple iterations of c in the general case. (In the example, however, they only correspond to a single iteration of c.) Thus, the residual transition paths are repeatable in c.

It should be obvious that for each residual transition path p'_k, $1 \leq k \leq r_1$, it follows that $\mathbf{cp}(p'_k) = \mathbf{ecp} - \mathbf{cp}(c) = \mathbf{cp} + \mathbf{cp}(c) - \mathbf{cp}(c) = \mathbf{cp}$. Thus, the static communication pattern **cp** is repeatable in c from mode m, i.e., repeatable($\mathbf{cp}, c, m$) = $\top$.

The other direction, i.e., given repeatable($\mathbf{cp}, c, m$) = $\top$ should be obvious from the above: It is always possible to traverse one or more transitions from m according to **cp** to reach a mode m' of c, then to perform one (or more) complete iterations of c (thereby returning to mode m'), from which **cp** is again repeatable, etc. □

Theorem 4.4 (Path compatibility)**.** Given a transition path p' which reaches a transition cycle $c = \langle t_1, \ldots, t_n \rangle$ in mode m_1 (cf. Figure 4.8). Note that p' may not be an elementary path in the general case. If the following condition is not satisfied, then it is not possible to successfully validate the FSM:

$$
\begin{aligned}
&(\text{repeatable}(\mathbf{cp}(p') &&, c, m_1) = \top) \vee \\
\vee\ &(\text{repeatable}(\mathbf{cp}(p'^\frown\langle t_1 \rangle)) &&, c, m_2) = \top) \vee \\
\vee\ &(\text{repeatable}(\mathbf{cp}(p'^\frown\langle t_1, t_2 \rangle)) &&, c, m_3) = \top) \vee \\
\vee\ &\ldots && \vee \\
\vee\ &(\text{repeatable}(\mathbf{cp}(p'^\frown\langle t_1, \ldots, t_{n-1} \rangle), c, m_n) = \top)
\end{aligned}
$$

In other words, $\mathbf{cp}(p')$ must be repeatable from m_1 in c, or $\mathbf{cp}(p'^\frown\langle t_1 \rangle)$ must be repeatable from m_2 in c, or $\mathbf{cp}(p'^\frown\langle t_1, t_2 \rangle)$ must be repeatable from m_3 in c, etc. Note that according to Theorem 4.3, no more transitions have to appended to p', because repeatable($\mathbf{cp}(p'^\frown\langle t_1, \ldots, t_n \rangle, c, m_1)$ = repeatable($\mathbf{cp}(p'^\frown c), c, m_1$) = repeatable($\mathbf{cp}(p') + \mathbf{cp}(c), c, m_1$) = repeatable($\mathbf{cp}(p'), c, m_1$).

Proof. Assume that Theorem 4.4 is not satisfied for a given path p' and cycle $c = \langle t_1, \ldots, t_n \rangle$, but that the FSM is successfully validated for a static communication pattern $\mathbf{cp} \geq \mathbf{cp}(p')$. Note that once a valid pattern has been found, it is always possible to enlarge it, i.e., if $\mathbf{cp}$ is a repeatable pattern, so is $q \cdot \mathbf{cp}, q > 0$. Then, because $\mathbf{cp} \geq \mathbf{cp}(p')$, it follows that $\mathbf{cp}$ can be written as $\mathbf{cp} = \mathbf{cp}(p') + r \cdot \mathbf{cp}(c) + \mathbf{cp}(p)$ such that $r \geq 0$, and $\mathbf{cp}(p) < \mathbf{cp}(c)$. Here, p denotes a (possibly empty) transition path from m_1 in c to a mode m_k in c (i.e., m_k denotes the target mode of the last transition of p). Note that this transition path must exist, due to $\mathbf{cp}$ being a validated repeatable pattern. Due to the same reason, it follows that $\text{repeatable}(\mathbf{cp}, c, m_k) = \top$. Then, applying Theorem 4.3, it follows that:

$$\begin{aligned}
\top &= \text{repeatable}(\mathbf{cp}, c, m_k) \\
&= \text{repeatable}(\mathbf{cp}(p') + r \cdot \mathbf{cp}(c) + \mathbf{cp}(p), c, m_k) \\
&= \text{repeatable}(\mathbf{cp}(p') + \mathbf{cp}(p), c, m_k) \\
&= \text{repeatable}(\mathbf{cp}(p'^\frown p), c, m_k)
\end{aligned}$$

However, $\text{repeatable}(\mathbf{cp}(p'^\frown p), c, m_k) = \top$ is a contradiction to the assumption that Theorem 4.4 is not satisfied for p' and cycle c. □

Note that it is not sufficient to validate all *elementary* paths from m_0 that reach a mode m traversed by a cycle c. Theorem 4.4 is therefore checked during the FSM traversal for all paths p' from m_0 that reach a cycle c. Obviously, paths p' which have already been validated can be cached to speed up the validation process. If a path p' is encountered which does not adhere to Theorem 4.4, validation fails ultimately. In this case, the FSM does not adhere to the static communication pattern $\mathbf{cp}$, or to any larger pattern which may be constructed from the selected reference path p_∞.

It should be noted that Theorem 4.4 implies that eventually, all transition cycles c are reachable. However, it is not sufficient to show that an elementary transition path p' from m_0 to c exists: Consider a port for which p' consumes or produces some tokens, but the selected reference path p_∞ does not. In this case, $\mathbf{cp}$ will always have a zero entry for this port, regardless of how many transitions of p_∞ are added to $\mathbf{cp}$. However, this situation is easily identified by analyzing the selected reference path, and validation aborted.

Finally, it can be observed that if all cycles are compatible according to Theorem 4.2, and no incompatible transition paths can be found during the validation procedure according to Theorem 4.4, the validation of the FSM is successfully completed eventually. The sketch of the proof is as follows: Assume that a non-elementary path p' first traverses a cycle c_1 before reaching another cycle $c_2 \neq c_1$ in mode m. According to Theorem 4.2, all cycles are compatible. Thus, we can find integers $r_1 > 0$ and $r_2 > 0$ such that $r_1 \, \mathbf{cp}(c_1) = r_2 \, \mathbf{cp}(c_2)$. Assume that p' traverses c_1 at least r_1 times, which is eventually the case when

the constructed communication patterns become large enough. In this case, r_1 iterations of c_1 can be eliminated from p', and the remaining shortened path p'' still reaches cycle c_2 in the same mode m. Now, appending r_2 iterations of c_2 to the shortened path p'', it follows that $\text{repeatable}(\mathbf{cp}(p'), c_2, m) = \text{repeatable}(\mathbf{cp}(p'') + r_2\,\mathbf{cp}(c_2), c_2, m) = \text{repeatable}(\mathbf{cp}(p''), c_2, m)$ according to Theorem 4.3. As the shortened path p'' must have been evaluated before for smaller communication patterns, validation would have already stopped earlier if p'' is not repeatable according to Theorem 4.4.

Example 4.9. Consider the FSM shown in Figure 4.7a. For example, transition path $p'_1 = \langle t_1 \rangle$ reaches the only cycle $c = \langle t_2, t_3, t_4, t_5 \rangle$. As $\text{repeatable}(\mathbf{cp}(p'_1 \frown \langle t_2 \rangle), c, m_1) = \top$, Theorem 4.4 is satisfied for p'_1. The other transition path $p'_2 = \langle t_6, t_7 \rangle$ also satisfies Theorem 4.4. As Theorem 4.2 is trivially satisfied, we can conclude that a repeatable communication pattern exists, which has already been shown in Example 4.6.

Consider the FSM shown in Figure 4.7b. Example 4.8 has already shown that Theorem 4.2 is satisfied. Assume that the selected reference path is $p_\infty = p \frown c_1 \frown c_1 \frown \ldots$, where $p = \langle t_1 \rangle$ and $c_1 = \langle t_2, t_3 \rangle$. Note that the other transition cycle is $c_2 = \langle t_6, t_7 \rangle$. In this case, the communication patterns constructed from p_∞ are $\mathbf{cp}_1 = (1, 1)$, $\mathbf{cp}_2 = (1, 3)$, $\mathbf{cp}_3 = (2, 4)$, $\mathbf{cp}_4 = (2, 6)$, $\mathbf{cp}_5 = (3, 7)$, $\mathbf{cp}_6 = (3, 9)$, and so on. Concerning the elementary paths into each cycle, all three paths $p'_1 = \langle t_1 \rangle$, $p'_2 = \langle t_1, t_2, t_4 \rangle$ and $p'_3 = \langle t_5 \rangle$ satisfy Theorem 4.4 for the corresponding transition cycle c_1 or c_2. However, traversing cycle c_1 once, it follows for the resulting non-elementary path $p'_4 = \langle t_1, t_2, t_3, t_2, t_4 \rangle$ that $\mathbf{cp}(p'_4) = (3, 9)$. It is easily verified that p'_4 does not satisfy Theorem 4.4 w.r.t. cycle c_2. Thus, after evaluating $\mathbf{cp}_6 \geq p'_4$, we can conclude that no repeatable (even larger) communication pattern for the FSM in Figure 4.7b exists.

Partitioning into Phases

It should be noted that, in principle, an identified repeatable static communication pattern $\mathbf{cp}$ could be partitioned into arbitrary phases. However, the more fine-grained the phases become, the smaller the chance becomes that the chosen partitioning is successfully validated, as each phase must be covered by distinct transition paths. In other words, a single transition cannot span multiple phases (whereas a single phase may span multiple transitions).

In this section, we propose a partitioning scheme such that the actor, when treated as a CSDF actor, does not introduce deadlocks into the model. To this end, we assume that we are given a transition path p from the initial mode m_0 in the validated actor FSM such that $\mathbf{cp}(p) = \mathbf{cp}$. This path is used to partition the communication pattern $\mathbf{cp}$ into phases.

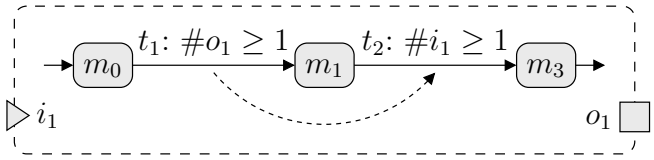
In order to illustrate the property of the proposed partitioning approach, assume that the given transition path p is already partitioned into $\tau \in \mathbb{N}$ subpaths such that $p = p_1 \frown \ldots \frown p_\tau$. Then, in order to prevent the introduction of deadlocks into the model, the transitions comprising each transition subpath p_k must consume all tokens before producing any tokens:

Definition 4.2. Given a transition path p which is partitioned into τ subpaths such that $p = p_1 \frown \ldots \frown p_\tau$. The partitioning is well-formed if for each subpath $p_k = \langle t_{k,1}, \ldots, t_{k,n} \rangle$ the following condition holds:

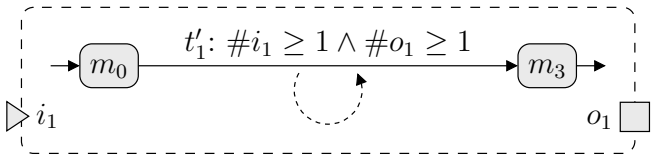
$$\exists j, 1 \leq j \leq n :$$
$$\left(\sum_{i=1}^{j} \mathbf{cons}(t_{k,i}) = \mathbf{cons}(p_k) \right) \wedge \left(\sum_{i=j}^{n} \mathbf{prod}(t_{k,i}) = \mathbf{prod}(p_k) \right)$$

Note that transition $t_{k,j}$ of a well-formed subpath p_k may consume and produce tokens. For the remaining transitions of p_k, this restriction stems from the possibility that tokens consumed by a transition t_2 may depend on tokens produced by a different transition t_1, either via self-loop FIFO channels, or via feedback loops over multiple actors. As the transitions which comprise a phase are executed atomically, such dependencies between transitions may become cyclic dependencies, thereby introducing deadlocks into the model. This restriction could be relaxed after the analysis of feedback loops.

Example 4.10. Consider Figure 4.10: Assuming that transition t_2 requires the token produced by t_1, merging transitions t_1 and t_2 introduces a deadlock into the model.



a) Token dependency between transition t_1 and t_2.



b) Cyclic dependency after the merging of transitions t_1 and t_2.

Figure 4.10: The merging of transitions (a) may create cyclic dependencies (b) which must be avoided in the general case.

Algorithm 4.2 Validation of the partitioning of **cp** into phases

```
1: procedure ValidateTransition(Transition t, Phase k, cons′, prod′)
2:     if cons(t) ≰ cons′ ∨ prod(t) ≰ prod′ then                ▷ Condition 4.1
3:         return ⊥
4:     end if
5:     if cons(t) < cons′ ∧ prod(t) ≠ 0 then                    ▷ Condition 4.2
6:         return ⊥
7:     end if
8:     cons′ ← cons′ − cons(t)
9:     prod′ ← prod′ − prod(t)
10:    if cons′ = 0 ∧ prod′ = 0 then                  ▷ Finished the current phase
11:        if k = τ then                                ▷ Process the next phase
12:            k ← 1
13:        else
14:            k ← k + 1
15:        end if
16:        cons′ ← cons(p_k)
17:        prod′ ← prod(p_k)
18:    end if
19:    ...                                  ▷ Process possible target modes of t
20: end procedure
```

Then, in order to partition **cp** into phases, we determine the longest prefix path p_1 of $p = \langle t_1, \ldots, t_i, t_j, \ldots, t_n \rangle$ which still satisfies Condition 4.1. Assuming that $p_1 = \langle t_1, \ldots, t_i \rangle$ is the longest prefix path which satisfies Condition 4.1, any longer prefix path $p' = \langle t_1, \ldots, t_j, \ldots \rangle$ therefore does not satisfy Condition 4.1. Subsequently, we determine the longest subpath p_2 of p (starting from transition t_j) which satisfies Condition 4.1, and so on. In this way, p is successively partitioned into τ phases such that $p = p_1 \frown \ldots \frown p_\tau$.

Finally, the FSM is traversed to verify that the partitioning of **cp** into phases is viable. Note that if validation fails for the computed phases, validation for any other partitioning of **cp** into phases which satisfies Condition 4.2 would also fail. This is due to the fact that the partitioning procedure creates as few phases as possible (but as many as necessary) for the given path p, and p is a transition path from m_0, i.e., p is also validated.

The validation of the chosen partitioning of **cp** into phases proceeds analogously to Algorithm 4.1. Thus, only the relevant procedure ValidateTransition is shown in Algorithm 4.2. In contrast to Algorithm 4.1, the additional parameter k identifies the current phase k, $1 \leq k \leq \tau$, for which transition t is validated. Note that $\mathbf{cons}'$ and $\mathbf{prod}'$ now denote the number of remaining tokens yet to be consumed and produced by subsequent transitions in the current phase k only.

Procedure ValidateTransition validates a transition t as follows: First, t must not consume or produce more tokens than remain for the current phase (cf. lines 2–4). Next, Condition 4.2 is checked. Condition 4.2 is violated if t produces some tokens but does not consume all remaining tokens of the current phase, i.e., $\mathbf{prod}(t) \neq \mathbf{0}$ and $\mathbf{cons}(t) < \mathbf{cons}'$ (cf. lines 5–7). If Condition 4.2 is violated, the FSM does not adhere to the chosen partitioning of **cp** into phases. Otherwise, $\mathbf{cons}'$ and $\mathbf{prod}'$ are updated (cf. lines 8–9). If no tokens remain, i.e., $\mathbf{cons}' = \mathbf{0}$ and $\mathbf{prod}' = \mathbf{0}$, transition t successfully completed the validation of the current phase k. In this case, $\mathbf{cons}'$ and $\mathbf{prod}'$ are reset to the the token consumption and production rates of the next phase (cf. lines 10–18). Remember that p_k denotes the transition path corresponding to phase k. Subsequently, the possible target modes of t are processed as done by Algorithm 4.1. Finally, the initial mode m_0 starts in phase 1, i.e., $k = 1$, $\mathbf{cons}' = \mathbf{cons}(p_1)$, and $\mathbf{prod}' = \mathbf{prod}(p_1)$, where p_1 denotes the transition path corresponding to phase 1.

Example 4.11. Consider the FSM shown in Figure 4.7a. As shown in Example 4.6, the smallest repeatable static communication pattern is given by $\mathbf{cp} = (2, 1, 1)$. A possible transition path p which can be used to compute the phases is $p = \langle t_6, t_7, t_5, t_2 \rangle$. In this case, the whole path p satisfies Condition 4.2, as only the last transition t_2 produces a token. Thus, we set $\tau = 1$, and $p_1 = p$. Obviously, validating the FSM against this partitioning scheme succeeds. Therefore, the actor FSM corresponds to a CSDF actor with one phase and the communication behavior as specified by **cons** and **prod**.

Merging of Transitions

Finally, we generate an actor in the proposed dataflow model which corresponds to the CSDF actor determined by the classification procedure. In the following, *underlying actor FSM* refers to the FSM of the actor identified as a CSDF actor (cf. Figure 4.7), while *(normalized) actor FSM* refers to the FSM of the transformed actor according to Figure 4.5. The structure of the normalized actor FSM has been described in Section 4.3.1. Concerning the action function f_k for a phase $k, 1 \leq k \leq \tau$, one possibility is to *quasi-statically* schedule the transitions of the underlying actor FSM. This requires enumeration of all possible transition paths, and adding guard function evaluations as necessary. To this end, we allocate a new actor variable m_u which reflects the current mode of the underlying actor FSM. Its initial value is the initial mode of the underlying FSM. In contrast, the initial mode of the normalized actor FSM is the mode allocated for the first phase. This variable is required because different transition paths of the underlying actor FSM may have to be executed depending on m_u in the

Algorithm 4.3 Action function corresponding to the CSDF actor derived from the actor FSM shown in Figure 4.7a

```
1: procedure f_1(..., v, m)
2:     assert(m = m_1)                    ▷ The CSDF actor has only one phase (τ = 1)
3:     Let v_orig⌢⟨m_u⟩ ← v
4:     switch m_u do
5:         case m_0
6:             if f_g1(..., m_0) = ⊤ then
7:                 (..., m_u) ← t_1.f_a(..., m_0)
8:                 (..., m_u) ← t_2.f_a(..., m_1)
9:             else
10:                assert(f_g2(..., m_0) = ⊤)          ▷ Requirement 4.1
11:                (..., m_u) ← t_6.f_a(..., m_0)
12:                (..., m_u) ← t_7.f_a(..., m_5)
13:                (..., m_u) ← t_5.f_a(..., m_4)
14:                (..., m_u) ← t_2.f_a(..., m_1)
15:            end if
16:            break
17:        case m_2
18:            (..., m_u) ← t_3.f_a(..., m_0)
19:            (..., m_u) ← t_4.f_a(..., m_5)
20:            (..., m_u) ← t_5.f_a(..., m_4)
21:            (..., m_u) ← t_2.f_a(..., m_1)
22:            break
23:    end switch
24:    return (..., v_orig⌢⟨m_u⟩, m_1)
25: end procedure
```

general case. Note that the paths can be constructed by additional bookkeeping logic in Algorithm 4.2.

Example 4.12. Consider the actor FSM in Figure 4.7a. As shown in Example 4.6, the communication behavior of the actor FSM adheres to the static communication behavior of an SDF actor with $\mathbf{cp} = (2, 1, 1)$. Thus, the normalized actor FSM consists of one mode m_1, and one self-loop transition t_1. The action function f_1 of t_1 is shown in Figure 4.3. For the sake of clarity, token consumption and production details have been omitted. Depending on the current mode of the underlying actor FSM, different transition paths are executed. In particular, in mode m_0, the guard function f_{g1} must be evaluated. Thus, f_1 implements a quasi-static schedule. Note that the guard function f_{g2} need not be evaluated if $f_{g1} = \bot$, as one of both transitions t_1 or t_6 of the underlying actor FSM must be enabled due to Requirement 4.1.

If the possible number of transition paths becomes very large, quasi-static scheduling may be no longer feasible. In this case, a *dynamic scheduling scheme* could be used, which dynamically schedules the transitions of the underlying actor FSM. To this end, the transformed actor FSM could allocate an additional actor variable for each input port and output port, which specify the number of tokens which must yet be consumed and produced in the current phase. Note that these variables correspond to the vectors $\mathbf{cons}'$ and $\mathbf{prod}'$ used in Algorithm 4.2. Then, we can execute transitions of the underlying FSM as long as $\mathbf{cons}' \neq \mathbf{0}$ or $\mathbf{prod}' \neq \mathbf{0}$. When $\mathbf{cons}' = \mathbf{0}$ and $\mathbf{prod}' = \mathbf{0}$, the current phase is finished, and $\mathbf{cons}'$ and $\mathbf{prod}'$ are initialized with the consumption and production rates of the next phase. Note that static analysis has shown that $\mathbf{cons}'$ and $\mathbf{prod}'$ will always become zero after some transitions have been executed. While this scheduling scheme induces some run-time overhead, it does not require the explicit construction of transition paths, thereby potentially reducing the code size of the synthesized actor.

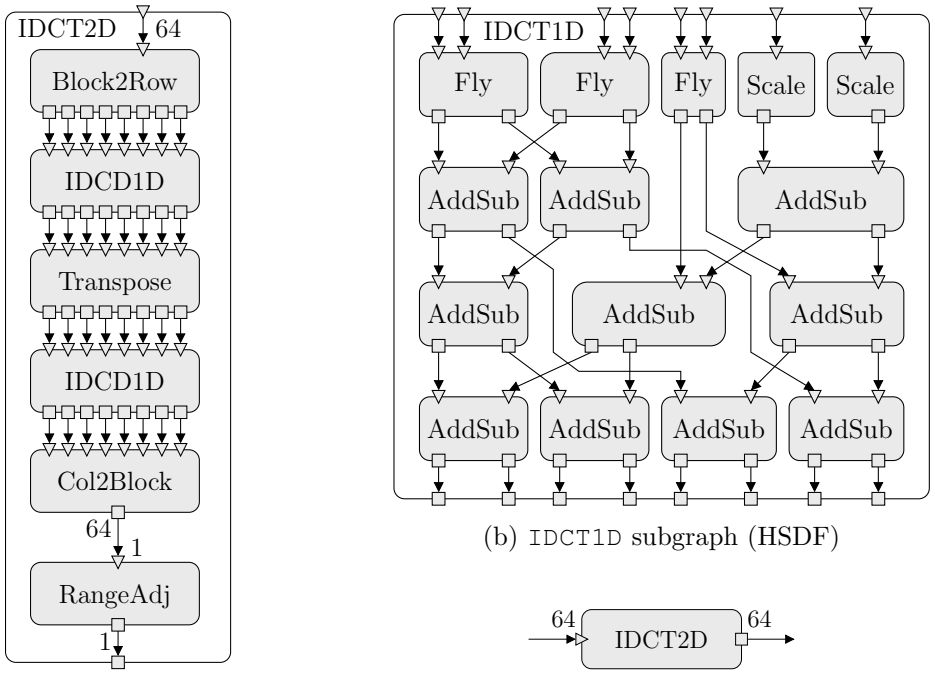
Due to the general undecidable nature of the problem, the presented classification algorithm represents only a sufficient but not a necessary condition for an actor to adhere to a static dataflow model. In particular, the classification algorithm lacks propagation of guard condition invariants across multiple transitions, i.e., even if we know that a certain guard function has been evaluated to true, and subsequent action functions do not change the guard value, the next evaluation of this guard function will again be considered uncertain.

Finally, the difference between the general actor model presented in Section 3.1 and the restricted actor model without guard functions described in Section 3.3 affects Requirement 4.1, which then only requires that eventually, enough tokens and free places must be available in order for any transition to be enabled. However, as it is unknown which of the possible target modes is selected by a transition at run time, the presented classification algorithm still only represents a sufficient but not a necessary condition for an actor to adhere to a static dataflow model.

4.3.3 Results

In order to evaluate the optimization potential for real-world examples, the classification algorithm has been applied to a JPEG decoder as seen in Figure 2.3. As a result, the `JPEGSource` actor, the `InverseQuant` actor, the `InverseZigZag` actor, and additionally all actors of the hierarchical `IDCT2D` actor are classified either into the SDF or CSDF MoCs. The hierarchical `IDCT2D` actor transforms blocks of 8×8 frequency coefficients into equally sized image blocks, and is depicted in detail in Figure 4.11a and b.

This enables the application of model-based optimizations including the clustering of the static dataflow actors into a single *composite actor* shown in



(a) hierarchical `IDCT2D` actor

(b) `IDCT1D` subgraph (HSDF)

(c) composite `IDCT2D` actor (SDF)

Figure 4.11: Composite `IDCT2D` actor (c) resulting from clustering all static dataflow actors of (a) and (b). Note that token consumption and production rates are equal to 1 unless indicated otherwise.

Figure 4.11c by statically scheduling all actors from Figure 4.11a and b. This composite `IDCT2D` actor must, however, be dynamically scheduled due to the heterogeneous environment in which it is embedded (cf. Figure 2.3 on page 12): To determine if the composite `IDCT2D` actor could be activated, both the incoming and outgoing channels must be checked for the required number of tokens and free places.

Without knowing the underlying model of computation, this optimization step would not be possible, i.e., no composite actor could be generated and the `IDCT2D` actor must be synthesized as seen in Figure 4.11a and b. Therefore, instead of having to check the fill level of only two channels, the dynamic scheduler

Hierarchical `IDCT2D` actor	Composite `IDCT2D` actor	Reduction
0.305s	0.132s	57%

Table 4.1: Measured latencies for the synthesized JPEG decoder from Figure 2.3 on page 12. The latency is here defined as the time needed to decode a single JPEG picture of the size 176×144 pixels.

must check all 55 channels in turn, thereby wasting precious computational power. Moreover, besides checking the fill level of the channels, a dynamic round-robin scheduler must also poll each actor individually to see if it can fire or not, due to the guard functions associated with the transitions of the actors.

Comparing the latencies of the JPEG decoder, the version containing the composite `IDCT2D` actor outperforms the version implementing the hierarchical `IDCT2D` actor approximately by a factor of two, as summarized by Table 4.1. Due to the single-processor scheduling, the throughput could only be improved by the same value of 57%.

4.4 Boolean Dataflow

The Boolean dataflow (BDF) model has been introduced in [Buc93]. BDF extends CSDF by data-dependent SWITCH and SELECT actors (cf. Figure 4.12). The SWITCH actor consists of one control input port, one data input port, and two data output ports. The SWITCH actor, when fired, consumes a Boolean control token from its control input port, a data token from its data input port, and, depending on the value of the control token, forwards the data token to one of its data output ports. Analogously, the SELECT actor consists of one control input port, two data input ports, and one data output port. The SELECT actor, when fired, consumes a Boolean control token from its control input port, and, depending on the value of the control token, consumes a data token from one of its data input ports and forwards it to the data output port.

In order to determine the repetition vector of a BDF graph, variables are introduced, representing the non-static token production rates of SWITCH actors and the non-static token consumption rates of SELECT actors. When solving the balance equations, constraints for these variables may be calculated. In order for a repetition vector to exist, these constraints must be met. However, in order to automatically show whether these constraints are met or not, one would have to prove that two streams of Boolean tokens are identical, which is undecidable in the general case. Thus, [Buc93] suggests to "add assertions to the graph that would explicitly provide the missing information".

Example 4.13. Consider the BDF graph shown in Figure 4.12. It consists of the SDF actors v_1, v_3, v_4, and v_6 with static token consumption and production rates, a SWITCH actor v_2 with non-static token production rates, and a SELECT actor v_5 with non-static token consumption rates. The variables denoting the non-static token production and consumption rates could be interpreted as probabilities, or more intuitively, as the actual number of tokens produced and consumed. For example, if $p_1 = 1$, v_2 produces one token on its upper output port, and no tokens on its lower output port. In contrast, if $p_1 = 0$, v_2 produces

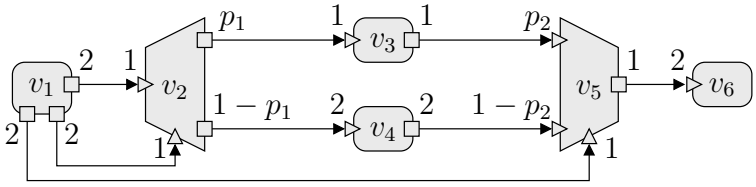


Figure 4.12: BDF graph with a SWITCH actor v_2 and a SELECT actor v_5.

no tokens on its upper output port, and one token on its lower output port. Calculating the repetition vector $\boldsymbol{\gamma}$, we first set $\gamma_1 = 1$. From the balance equations, it follows that $\gamma_2 = 2$. For v_3, it follows that $\gamma_3 = 2p_1$, while for v_4, it follows that $\gamma_4 = 1 - p_1$. For v_5, two values for γ_5 can be derived, based on γ_3 and γ_4: $\gamma'_{5,1} = 2 \cdot \frac{p_1}{p_2}$ and $\gamma'_{5,2} = 2 \cdot \frac{1-p_1}{1-p_2}$. Thus, in order for γ_5 to exist, $\gamma'_{5,1} = \gamma'_{5,2} \leftrightarrow p_1 = p_2$. Basically, this means that the streams of Boolean control tokens produced by v_1 must be equal, which, however, cannot be inferred from the BDF graph automatically. Assuming that $p_1 = p_2$, it follows that $\gamma_5 = 2$. Finally, for v_6, it follows that $\gamma_6 = 1$. Thus, the repetition vector $\boldsymbol{\gamma}$, which now depends on the variable p_1, evaluates to $\boldsymbol{\gamma}(p_1) = (1, 2, 2p_1, 1 - p_1, 2, 1)$. In order to determine a *complete cycle* of a BDF graph, further analysis of $\boldsymbol{\gamma}$ is necessary (cf. [Buc93]). Suffice to say, in CSDF graphs, a complete cycle corresponds to the number of actor firings specified by $\boldsymbol{\gamma}$. In contrast, for the example, a complete cycle is finished only when v_1 has produced an even number of tokens with value $\top$ or $\bot$, respectively. For example, a sequence $\langle \top, \top \rangle$ of produced control tokens corresponds to a complete cycle, as does the sequence $\langle \bot, \bot \rangle$. On the other hand, a sequence $\langle \top, \bot, \bot, \ldots \rangle$ forms a complete cycle only when a second control token with value $\top$ is produced, and the number of produced control tokens with value $\bot$ is even at this point.

4.4.1 Representation

BDF extends CSDF models by SWITCH and SELECT actors (cf. Figure 4.12). Figure 4.13 shows how these actor could be represented in the proposed dataflow model. For example, the SWITCH actor consisting of two input ports and two output ports can be implemented by means of the general actor model as shown in Figure 4.13a. The actor FSM consists of a single mode with two self-loop transitions t_1 and t_2. While both transitions consume a token from the control input port i_c and the data input port i_1, transition t_1 forwards the data token to output port o_1, and transition t_2 forwards the data token to output port o_2. The SELECT actor can be implemented analogously in the general actor model (cf. Figure 4.13c).

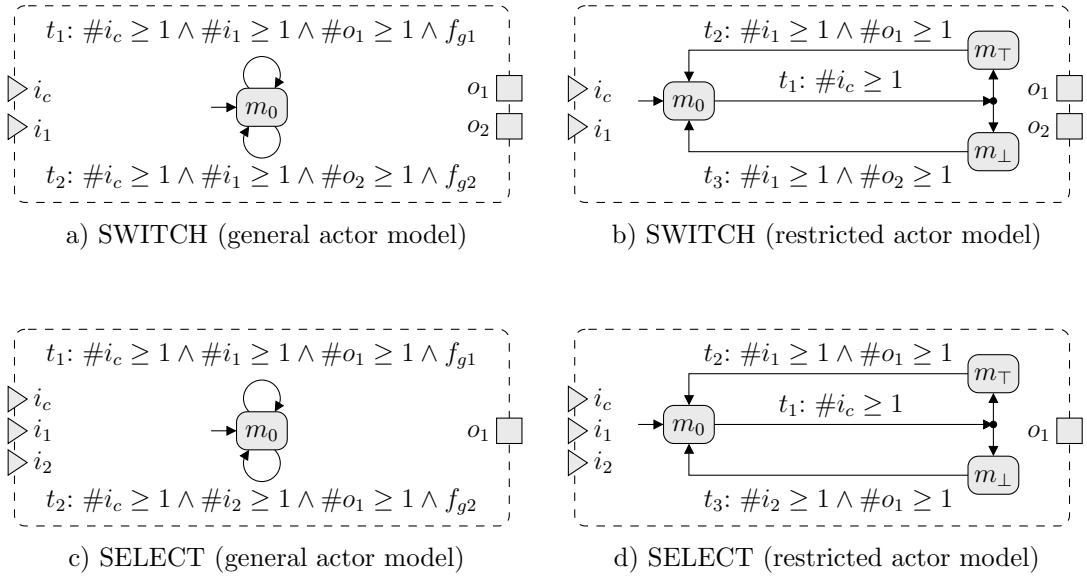


Figure 4.13: FSMs corresponding to the BDF SWITCH and SELECT actors in the proposed dataflow model.

In the restricted actor model, the SWITCH actor can be implemented as shown in Figure 4.13b: First, transition t_1 consumes a token from the control input port i_c, and selects either target mode $m_\top$ or $m_\bot$ depending on the value of the control token. Subsequently, one token is forwarded from the data input port i_1 to the corresponding output port o_1 or o_2. The SELECT actor can be implemented analogously in the restricted actor model (cf. Figure 4.13d).

4.4.2 Identification

Concerning the identification of SWITCH and SELECT actors, the token consumption and production rates are known (in contrast to CSDF actors as discussed in Section 4.3.2). For SWITCH actors, an actor must have two input ports and two output ports. Then, one of the input ports is assumed to be the control input port i_c. For each token consumed from i_c, exactly one token must be consumed from the other input port i_1, and exactly one token must be produced on exactly one of the output ports o_1 or o_2. In order to validate such token consumption and production patterns, Algorithm 4.1 can be modified to permit *alternative* communication patterns $\mathbf{cp}_k$.

For each alternative static communication pattern $\mathbf{cp}_k$, Algorithm 4.1 individually tracks the number of tokens yet to be consumed or produced by subsequent transitions, and excludes an alternative pattern if a transition t does not consume or produce less or equal tokens than specified by the value of $\mathbf{cp}'_k$. In this case, validation fails only when all alternative patterns have been excluded. If any $\mathbf{cp}'_k$

becomes zero, validation succeeds for the current transition path, and the next iteration is considered. In this case, all alternative patterns are considered again.

For a SWITCH actor as defined by BDF, two possible alternative patterns exist, namely $\mathbf{cp}_a = (1, 1, 1, 0)$, and $\mathbf{cp}_b = (1, 1, 0, 1)$. (Note that the first entry corresponds to the control input port i_c.) In other words, while no alternative token consumption rates are possible ($\mathbf{cons}_a = \mathbf{cons}_b = (1, 1)$), a token could either be produced on output port o_1 ($\mathbf{prod}_a = (1, 0)$), or on output port o_2 ($\mathbf{prod}_b = (0, 1)$). Note that i_c and i_1 are interchangeable w.r.t. the validation algorithm, i.e., it cannot be determined which input port is the control input port and which one is the data input port. This would require analysis of the guard function or action function, but as this analysis is undecidable in the general case, it is not considered here.

For a SELECT actor as defined by BDF, two possible alternative patterns exist, namely $\mathbf{cp}_a = (1, 1, 0, 1)$, and $\mathbf{cp}_b = (1, 0, 1, 1)$. (Note that the first entry corresponds to the control input port i_c.) In other words, while no alternative token production rates are possible ($\mathbf{prod}_a = \mathbf{prod}_b = (1)$), a token could either be consumed from input port i_1 ($\mathbf{cons}_a = (1, 1, 0)$), or from input port i_2 ($\mathbf{cons}_b = (1, 0, 1)$). In this case, it is possible to determine which input port corresponds to the control input port by validating the FSM for each input port assumed to be the control input port in turn. It should be obvious that validation succeeds only if the proper input port has been assumed to be the control input port.

Example 4.14. For the SWITCH actor in Figure 4.13a, the validation algorithm starts in mode m_0 with $\mathbf{cp}'_a = (1, 1, 1, 0)$, and $\mathbf{cp}'_b = (1, 1, 0, 1)$. Selecting transition t_1, $\mathbf{cp}(t_1) = (1, 1, 1, 0) \leq \mathbf{cp}'_a$, but $\mathbf{cp}(t_1) \not\leq \mathbf{cp}'_b$. Thus, alternative pattern $\mathbf{cp}'_b$ is excluded. For the remaining pattern, it follows that $\mathbf{cp}'_a \leftarrow \mathbf{cp}'_a - \mathbf{cp}(t_1) = (0, 0, 0, 0)$. As $\mathbf{cp}'_a = \mathbf{0}$, validation succeeds for transition t_1. Analogously, validation succeeds for transition t_2. Note that in this case, however, alternative pattern $\mathbf{cp}'_a$ is excluded. Thus, validation succeeds for all possible paths from m_0, and the actor under consideration indeed corresponds to a SWITCH actor. As outlined above, it cannot be determined which input port corresponds to the control input port. The alternative implementation of the SWITCH actor (cf. Figure 4.13b) is successfully validated analogously.

For the SELECT actor in Figure 4.13c, the validation algorithm starts in mode m_0 with $\mathbf{cp}'_a = (1, 1, 0, 1)$, and $\mathbf{cp}'_b = (1, 0, 1, 1)$. If the proper input port has been assumed to be the control input port, the FSM is successfully validated analogously to the SWITCH actor. If the wrong input port has been assumed to be the control input port, it is easily verified that either transition t_1 or transition t_2 fails validation. The alternative implementation of the SELECT actor (cf. Figure 4.13d) is successfully validated analogously.

In principle, the transformation of the FSM of an actor identified as a SWITCH or SELECT actor into a normalized representation can be performed as described for CSDF actors in Section 4.3.2. Thus, it is not described in more detail at this point. It should be noted that the alternative token consumption and production rates must be considered appropriately.

4.5 Parameterized Synchronous Dataflow

The parameterized synchronous dataflow (PSDF) model has been introduced in [BB00b]. Despite the name, the PSDF model is actually a *meta-modeling technique* which can be applied to any underlying DFG which has a well-defined notion of *iteration*. For the dataflow models reviewed in this section, this applies to the HSDF, SDF, CSDF, and BDF models. For the former three models, an *iteration* consists of the actor firings according to the repetition vector, while for the BDF model, an iteration consists of a complete cycle (cf. Example 4.13). Thus, we place PSDF above BDF in terms of expressiveness (cf. Figure 4.2).

A PSDF graph Φ consists of a *body* DFG Φ_b where token production and consumption rates may depend on some parameters, an *init* DFG Φ_i, and a *subinit* DFG Φ_s. The operational semantics of a PSDF graph Φ can be sketched as follows: First, the init graph Φ_i is executed once, which must configure those parameters of the body graph which must be known in order to calculate the repetition vector $\boldsymbol{\gamma}$. The values of these parameters are fixed for one iteration of the body graph Φ_b. Subsequently, the repetition vector of the body graph Φ_b is computed, followed by the computation of a schedule for Φ_b. Finally, one iteration of Φ_b is executed. Here, prior to the firing of an actor of Φ_b according to the schedule, the subinit graph Φ_s is executed, which (re-)configures the values of those parameters which are not set by Φ_i. For a detailed discussion of the subinit graph, we refer the reader to [BB00b].

Example 4.15. Consider the body graph Φ_b of a PSDF graph Φ shown in Figure 4.14. It consists of actors v_1, v_2, and v_3 whose token consumption and production rates depend upon parameters p_1 and p_2. The repetition vector evaluates to $\boldsymbol{\gamma} = (p_1, 1, 1)$. Thus, a possible schedule is $S = \langle v_1^{p_1}, v_2, v_3 \rangle$. As $\boldsymbol{\gamma}$ (and therefore also S) depend on the value of p_1, p_1 must be configured by the init graph Φ_i. In contrast, p_2 can be configured by the subinit graph Φ_s.

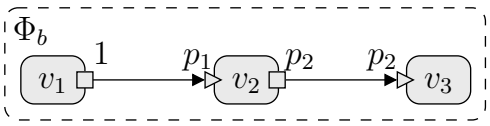


Figure 4.14: Body graph Φ_b of a PSDF graph Φ with parameters p_1 and p_2

The discussion of the model representation and analysis is postponed to Section 4.7.

4.6 Heterochronous Dataflow

The heterochronous dataflow (HDF) model has been introduced in [GLL99]. Similarly to PSDF, the token consumption and production rates of actors in a DFG can be reconfigured only after an iteration of the whole DFG is finished. While PSDF uses initialization DFGs for this purpose, HDF employs an FSM for this task. Moreover, while PSDF allows arbitrary configurations of parameters, HDF provides for a finite number of (static) *type signatures* per actor. Thus, while PSDF requires to compute the repetition vector and a schedule at run time after the token production and consumption values have been configured, HDF allows the same tasks to be performed at compile-time. However, if the number of possible combinations of type signatures becomes very large, these tasks could also be performed at run time.

In this context, it should be noted that the repetition vector of CSDF graphs can be calculated with a time complexity of $O(|V| + |E|)$, while a schedule for (acyclic) CSDF graphs can be found by topologically sorting the actors, which has the same linear time complexity. For cyclic graphs, however, scheduling is more difficult, and, in the worst case, may require a symbolic simulation of the DFG [HB07]. The discussion of the model representation and analysis is postponed to the next section.

4.7 Scenario-Aware Dataflow

The (general) scenario-aware dataflow (SADF) model has been introduced in [TGB+06]. Similar to the PSDF and HDF models, it extends SDF graphs by variable token consumption and production rates. To this end, each actor has an associated non-empty set of *scenarios*, each of which specifies the token consumption and production rates of the actor. Furthermore, actors are partitioned into *kernel* actors and *(scenario) detector* actors. While kernels represent the data processing part of a streaming application, detectors model the control part of the application that dynamically detects scenarios. Thus, a detector, when fired, determines the scenario in which itself and its depending kernels operate based on a stochastic model, and sends control tokens to the depending kernel actors accordingly.

In the general SADF model, scenarios can be changed within an iteration of the DFG. While this makes the model more expressive compared to the PSDF/HDF models, analyzability is reduced. In particular, static scheduling is not possible.

A restricted form of SADF is based on an FSM controlling the scenarios of the actors. This restricted form, however, is syntactically equivalent to the HDF model [SGTB11].

4.7.1 Representation

The PSDF model and the (general) SADF model allow token rates to be determined by the actors of the model, which is currently not supported by the proposed dataflow model, as it only provides for static token consumption and production values of actors in order to increase analyzability.

For the HDF model and the FSM-based SADF model, each actor (or the whole DFG) is associated with a finite set of static token production and consumption values. The selection of a specific set is governed by an FSM in both models. Thus, in order to implement such a behavior in the proposed dataflow model, the actors (or to be more precise, the transitions of the actors) could be duplicated for each set of token production and consumption values. As will be seen in Section 5.1, a hierarchical actor could then implement the switching of token production and consumption values by selecting the appropriate instance from the duplicated transitions. Note that the problem of scheduling the actors is an orthogonal problem, and is not considered here. Suffice to say, Section 5.3 shows that the proposed dataflow model allows a wide range of scheduling schemes to be implemented by action functions of hierarchical actors.

4.7.2 Identification

In contrast to CSDF and BDF models, HDF and FSM-based SADF models are represented by hierarchical actors in the proposed dataflow model. Thus, the identification of an analyzable HDF or FSM-based SADF model is more difficult. In particular, due to the implementation-oriented nature of the proposed dataflow model, action functions are considered to be black boxes which are only analyzable w.r.t. their input/output behavior (cf. Definition 3.4). While action functions therefore can be used to implement a wide range of scheduling schemes, it is not possible in the general case to extract a specific scheduling scheme from a given action function of a hierarchical actor. However, this information would be required to reconstruct the underlying parameterized SDF actors of an HDF or FSM-based SADF model.

4.8 Deterministic Dataflow

A DFG is *deterministic* if actors always produce the same token sequences, regardless of the order in which actors are fired. Here, the Kahn process network

(KPN) model, introduced in [Kah74], is widely seen as the foundation for other dataflow models. A simple language for parallel programming, the behavior of actors can be summarized as follows:

- Actors communicate only via FIFO channels of unbounded size.
- An actor either waits for data from exactly one input port (i.e., performs a blocking read on an input port), or is performing some computation. However, this blocking read requirement can be relaxed while still retaining deterministic behavior as shown in the following. Thus, blocking reads represent a sufficient but not a necessary condition in order for an actor to be deterministic.

More formally, a *Kahn process* is a function $\mathrm{F}\colon S^m \to S^n$ which maps a tuple of input sequences $s = (s_1, \ldots, s_m)$ into a tuple of output sequences $s' = (s_1, \ldots, s_n)$. A sequence $s_1 = \langle x_1, x_2 \rangle \in S$ is a *prefix* of a sequence $s_2 = \langle x_1, x_2, x_3 \rangle \in S$, written $s_1 \sqsubseteq s_1$. The empty sequence $\langle\rangle \in S$ is a prefix of any other sequence $s \in S$, i.e., $\langle\rangle \sqsubseteq s$. Note that sequences may be infinite, in which case they are called *streams*. The partial order "$\sqsubseteq$" over the set of finite and infinite sequences is a *complete partial order* (cpo) $(S, \sqsubseteq)$. The cpo $(S, \sqsubseteq)$ can be extended to tuples of sequences by pointwise comparison, i.e., $s = (s_1, \ldots, s_n) \sqsubseteq s' = (s'_1, \ldots, s'_n)$ if $\forall i, 1 \leq i \leq n : s_i \sqsubseteq s'_i$.

In order to be a Kahn process, the function F must be *continuous*. Informally, *continuous* functions prevent an actor to produce some output only after receiving an infinite amount of input. It can be shown [Kah74; LP95] that continuous functions are also *monotonic*, i.e., given two sequences $s_1 \sqsubseteq s_2$, it follows that $F(s_1) \sqsubseteq F(s_2)$. Informally, *monotonic* functions may compute parts of the output sequences given a prefix of the final input sequences. A well-known fixed-point theorem states that continuous functions defined on a complete partial order have a *least fixed-point* $s \in S^m$ such that $F(s) = s$, and for any other $s' \in S^m$ with $F(s') = s'$, it follows that $s \sqsubseteq s'$. Thus, in order to find s, one can start with the $m-$tuple of empty sequences $s_0 = \langle\rangle_m = (\langle\rangle, \ldots, \langle\rangle)$, and subsequently enlarge the sequences according to $s_{i+1} = F(s_i)$, until $s_{i+1} = s_i$.

Note that the procedure to find the least fixed-point is usually not a feasible execution strategy, because if an infinite sequence is generated by a process, this process will obviously never terminate. Thus, in practice, processes must be partially evaluated. This partial evaluation corresponds to the execution of firing rules on which many dataflow models are based, including the proposed dataflow model. Let $R \subseteq S^m$ be a set of *finite* m-tuples, representing the set of firing rules of an actor. Furthermore, the *firing function* $\mathrm{f}\colon R \to S^n$ specifies the *finite* token sequences produced by each firing rule $r \in R$.

In [Lee97; BHLP09], some criteria are described which must be satisfied by R and f such that a Kahn process F can be constructed from R and f. These rules

are stated as follows: Given two rules $r, r' \in R$ with $r \neq r'$ that have a common upper bound in S^m, i.e., $\exists s \in S^m : r \sqsubseteq s \wedge r' \sqsubseteq s$. Note that these rules r and r' are both enabled for such a tuple of input sequences s. Then, the order of execution of r and r' must make no difference w.r.t. the overall token sequences produced, i.e.[5], $\mathrm{f}(r)^\frown \mathrm{f}(r') = \mathrm{f}(r')^\frown \mathrm{f}(r)$, and r and r' do not have a common prefix other than the m-tuple of empty sequences, i.e., $\nexists s \in S^m, s \neq \langle\rangle_m : s \sqsubseteq r \wedge s \sqsubseteq r'$.

For the resulting definition of the Kahn process F based on R and f, we refer the reader to [Lee97].

Example 4.16. In the following, we consider token sequences S over an alphabet of $\{0, 1\}$, i.e., the token sequences are (possibly empty) binary strings. Then, the set of firing rules of an actor with one input port (i.e., $m = 1$) and one output port (i.e., $n = 1$) which simply forwards the tokens from the input port to the output port is as follows: $R = \{r_1, r_2\}$, with $r_1 = (\langle 0\rangle)$, $r_2 = (\langle 1\rangle)$, and $\forall r \in R : \mathrm{f}(r) = r$. Note that both firing rules r_1 and r_2 do not have a common upper bound in S. Thus, the process induced by R is continuous and therefore deterministic.

Consider the set of firing rules of an actor with two input ports (i.e., $m = 2$) and two output ports (i.e., $n = 2$) which forwards the tokens from input port i_1 to output port o_1, and the tokens from input port i_2 to output port o_2. Note that such an actor can be seen as the composition of two simple forwarding actors as in the previous example. Then, the set of firing rules is as follows: $R = \{r_1, r_2, r_3, r_4\}$, with $r_1 = (\langle 0\rangle, \langle\rangle)$, $r_2 = (\langle 1\rangle, \langle\rangle)$, $r_3 = (\langle\rangle, \langle 0\rangle)$, $r_4 = (\langle\rangle, \langle 1\rangle)$, and $\forall r \in R : \mathrm{f}(r) = r$. For example, rules r_1 and r_4 have a common upper bound in S^2, e.g., $(\langle 0\rangle, \langle 1\rangle)$. In this case, both rules are enabled. Thus, we have to show that $\mathrm{f}(r_1)^\frown \mathrm{f}(r_4) = \mathrm{f}(r_4)^\frown \mathrm{f}(r_1)$. Indeed, $\mathrm{f}(r_1)^\frown \mathrm{f}(r_4) = r_1{}^\frown r_4 = (\langle 0\rangle, \langle 1\rangle) = r_4{}^\frown r_1 = \mathrm{f}(r_4)^\frown \mathrm{f}(r_1)$. The remaining rules are verified analogously. Thus, this forwarding actor is also deterministic. Interestingly, however, it cannot be implemented by a blocking read on either input port: Assume that the actor blocks on i_1, and no more tokens arrive on i_1. Then, tokens on i_2 will never be forwarded.

Next, consider the set of firing rules of an actor with two input ports (i.e., $m = 2$) and one output port (i.e., $n = 1$), which forwards the tokens from i_1 and i_2 to the single output port o_1. The forwarding is solely based on the availability of tokens. Then, the set of firing rules is as follows: $R = \{r_1, r_2, r_3, r_4\}$, with $r_1 = (\langle 0\rangle, \langle\rangle)$, $r_2 = (\langle 1\rangle, \langle\rangle)$, $r_3 = (\langle\rangle, \langle 0\rangle)$, $r_4 = (\langle\rangle, \langle 1\rangle)$. Note that R corresponds to the set of firing rules from the previous example. However, the definition of f is different: $\mathrm{f}(r_1) = \mathrm{f}(r_3) = (\langle 0\rangle)$, and $\mathrm{f}(r_2) = \mathrm{f}(r_4) = (\langle 1\rangle)$. We

[5] The sequence concatenation operator "$^\frown$" is pointwise extended to tuples of sequences: $(a_1, \ldots, a_n)^\frown (b_1, \ldots, b_n) = (a_1{}^\frown b_1, \ldots a_n{}^\frown b_n)$ where a_i and b_i are sequences (of possibly different length).

consider again rules r_1 and r_4 which have a common upper bound in S^2. Then, $\mathrm{f}(r_1)^\frown \mathrm{f}(r_4) = (\langle 0\rangle)^\frown(\langle 1\rangle) = (\langle 0,1\rangle) \neq (\langle 1,0\rangle) = (\langle 1\rangle)^\frown(\langle 0\rangle) = \mathrm{f}(r_4)^\frown \mathrm{f}(r_1)$. Thus, this merge actor is nondeterministic (hence the name *nondeterministic merge*), as firing r_1 after r_4 yields a different result than firing r_1 before r_4.

In contrast, the SWITCH and SELECT actors of the BDF model (cf. Section 4.4) are deterministic: For the SELECT actor, the firing rules are as follows: $R = \{r_1, r_2, r_3, r_4\}$, with $r_1 = (\langle 0\rangle, \langle 0\rangle, \langle\rangle)$, $r_2 = (\langle 0\rangle, \langle 1\rangle, \langle\rangle)$, $r_3 = (\langle 1\rangle, \langle\rangle, \langle 0\rangle)$, $r_4 = (\langle 1\rangle, \langle\rangle, \langle 1\rangle)$, and $\mathrm{f}(r_1) = \mathrm{f}(r_3) = (\langle 0\rangle)$, and $\mathrm{f}(r_2) = \mathrm{f}(r_4) = (\langle 1\rangle)$. Note that compared to the nondeterministic merge, each firing rule consumes an additional token from the control input port. As no pair of rules has a common upper bound in S^3, we can conclude that the SELECT actor is deterministic (and, analogously, the SWITCH actor as well).

4.8.1 Representation

As the proposed dataflow model is based on transitions, a set of (deterministic) firing rules R and the associated firing function f can be easily transformed into the proposed dataflow model as follows: A single mode m_0 is allocated which also corresponds to the initial mode. For each firing rule $r \in R$, a transition $t = (m, M', f_g, f_a)$ is allocated such that $m = m_0$, and $M' = \{m_0\}$. Note that as firing rules are not governed by an FSM, the normalized representation outlined in this section simply allocates a single actor mode m_0 and attaches all firing rules as self-loop transitions to m_0. In other words, all firing rules are assumed to be potentially enabled at the same time. Other representations are conceivable, but would require to infer an FSM from the firing rules such that no deadlocks are introduced into the model. This could be part of future work. Concerning the action function f_a, the number of tokens consumed from each input port $p \in I$ is set to the length of the corresponding token sequence of the m-tuple r, while the number of tokens produced on each output port $p \in O$ is set to the length of the corresponding token sequence of the n-tuple $\mathrm{f}(r)$. The behavior of the action function corresponds to $\mathrm{f}(r)$, which specifies the values of the produced tokens. Concerning the guard function f_g, the values of peek are set to the values of cons, and f_g basically compares the values of the peeked tokens against the values specified by r.

4.8.2 Identification

As explained in the previous section, firing rules are evaluated by guard functions in the proposed dataflow model. While guard functions therefore can be used to implement a wide range of predicates, it is not possible in the general case to extract the values against which the peeked tokens are compared from a given

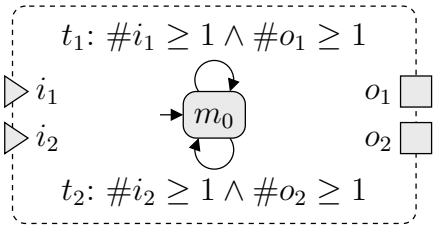


Figure 4.15: Deterministic forwarding actor.

guard function, which would be required in order to infer the firing rules as described above. Moreover, as action functions are also considered to be black boxes, the values of produced tokens cannot be inferred from an action function in the general case.

Thus, we have to abstract from the token values specified by the set of firing rules R and the firing function f. To this end, we use the token value "$*$", which matches any token (but not any *token sequence*). Note that the tokens produced according to the firing function f are indexed by the firing rule (or transition) which produces them. For example, assume that firing rule r_i produces a token $*_i$, while firing rule r_j produces a token $*_j$. Then, if $i \neq j$, both tokens are considered to have a different value.

Note that this analysis applies only to functional actors, i.e., for actors with no variables and only one mode m_0. In the general case, however, an actor in the proposed dataflow model may have state, i.e., variables and multiple modes. For the purpose of this analysis, the actor state must be explicitly modeled by one (or more) self-loop FIFO channels (cf. [LP95]). In the general case, it is not known statically how the current actor state is transformed by an action function, as these transformations may be data-dependent. Thus, the tokens representing the current actor state are also specified by means of the token value "$*$" in the general case. However, concerning the actor mode, some static knowledge may be exploited as for a given transition t, the set of possible target modes M' is known at compile time. In particular, if M' consists only of a single mode, i.e., $M' = \{m'\}$, m' must be selected as target mode by the action function $t.f_a$. Note that firing rules are represented by stateless actors in the proposed dataflow model as described in the previous section.

Finally, guard functions and action functions in the proposed dataflow model are assumed to be deterministic in the sense that output tokens and the transformed actor state only depend on input tokens and the actor state.

Example 4.17. Consider the actor FSM shown in Figure 4.15, which forwards tokens from input port i_1 to output port o_1, and tokens from input port i_2 to output port o_2, possibly transforming the token values in the process. In this case, the actor is assumed to be stateless, i.e., the set of actor variables

is empty, and the set of modes consists of the single mode m_0. Thus, the set of firing rules derived from the FSM is as follows: $R = \{r_1, r_2\}$, with $r_1 = (\langle * \rangle, \langle\rangle, \langle m_0 \rangle)$, $r_2 = (\langle\rangle, \langle * \rangle, \langle m_0 \rangle)$, $\mathrm{f}(r_1) = (\langle *_1 \rangle, \langle\rangle, \langle m_0 \rangle)$, and $\mathrm{f}(r_2) = (\langle *_2 \rangle, \langle\rangle, \langle m_0 \rangle)$. Note that each rule matches any token, but the produced tokens have been tagged by the firing rule which produces them. Additionally, the actor mode is explicitly represented by R and f. For example, rules r_1 and r_2 have a common upper bound in S^3, e.g., $(\langle * \rangle, \langle * \rangle, \langle m_0 \rangle)$. In this case, both rules are enabled. Thus, we have to show that $\mathrm{f}(r_1)^\frown \mathrm{f}(r_2) = \mathrm{f}(r_2)^\frown \mathrm{f}(r_1)$. Indeed, $\mathrm{f}(r_1)^\frown \mathrm{f}(r_2) = (\langle *_1 \rangle, \langle *_2 \rangle, \langle m_0, m_0 \rangle) = \mathrm{f}(r_2)^\frown \mathrm{f}(r_1)$. Thus, this forwarding actor shows deterministic behavior (under the assumption that the action functions are deterministic).

Next, consider again the set of firing rules of a nondeterministic merge actor with two input ports (i.e., $m = 2$) and one output port (i.e., $n = 1$), which forwards the tokens from i_1 and i_2 to the single output port o_1. Here, the set of firing rules derived from the corresponding actor FSM (not shown) is as follows: $R = \{r_1, r_2\}$, with $r_1 = (\langle * \rangle, \langle\rangle, \langle m_0 \rangle)$, $r_2 = (\langle\rangle, \langle * \rangle, \langle m_0 \rangle)$. Note that R corresponds to the set of firing rules from the previous example. However, the definition of f is different: $\mathrm{f}(r_1) = (\langle *_1 \rangle, \langle m_0 \rangle)$, and $\mathrm{f}(r_2) = (\langle *_2 \rangle, \langle m_0 \rangle)$. We consider again rules r_1 and r_2 which have a common upper bound in S^3. Then, $\mathrm{f}(r_1)^\frown \mathrm{f}(r_2) = (\langle *_1, *_2 \rangle, \langle m_0, m_0 \rangle) \neq (\langle *_2, *_1 \rangle, \langle m_0, m_0 \rangle) = \mathrm{f}(r_2)^\frown \mathrm{f}(r_1)$. Thus, this merge actor is correctly analyzed to show nondeterministic behavior, as firing r_1 after r_2 yields a different result than firing r_1 before r_2.

For the SELECT actor of the BDF model as shown in Figure 4.13c, the firing rules derived from the actor FSM are as follows: $R = \{r_1, r_2\}$, with $r_1 = (\langle * \rangle, \langle * \rangle, \langle\rangle, \langle m_0 \rangle)$, $r_2 = (\langle * \rangle, \langle\rangle, \langle * \rangle, \langle m_0 \rangle)$, $\mathrm{f}(r_1) = (\langle *_1 \rangle, \langle m_0 \rangle)$, and $\mathrm{f}(r_2) = (\langle *_2 \rangle, \langle m_0 \rangle)$. In this case, there is a common upper bound in S^4, e.g., $(\langle * \rangle, \langle * \rangle, \langle * \rangle, \langle m_0 \rangle)$. It follows that $\mathrm{f}(r_1)^\frown \mathrm{f}(r_2) = (\langle *_1, *_2 \rangle, \langle m_0, m_0 \rangle) \neq (\langle *_2, *_1 \rangle, \langle m_0, m_0 \rangle) = \mathrm{f}(r_2)^\frown \mathrm{f}(r_1)$. Thus, in this case, the SELECT actor would be classified as nondeterministic, even though transitions t_1 and t_2 cannot be enabled at the same time. However, due to the black box approach for guard functions, this property is undecidable in the general case.

In contrast, consider the SELECT actor implemented by means of the restricted actor model (cf. Figure 4.13d). In this case, the result of the guard function is explicitly represented by the distinct modes $m_\top$ and $m_\bot$. The firing rules derived from the actor FSM are as follows: $R = \{r_1, r_2, r_3\}$, with $r_1 = (\langle * \rangle, \langle\rangle, \langle\rangle, \langle m_0 \rangle)$, $r_2 = (\langle\rangle, \langle * \rangle, \langle\rangle, \langle m_\top \rangle)$, and $r_3 = (\langle\rangle, \langle\rangle, \langle * \rangle, \langle m_\bot \rangle)$. In this case, the firing rules do not have a common upper bound in S^4. In particular, it is important to understand that $\langle * \rangle$ is *not* an upper bound of $\langle m_0 \rangle$, $\langle m_\top \rangle$, and $\langle m_\bot \rangle$. Thus, the SELECT actor implemented by means of the restricted actor model is correctly analyzed to show deterministic behavior. This example shows that the restricted actor model may improve the analyzability of actors.

4.9 Nondeterministic Dataflow

A DFG is *nondeterministic* if the token sequences produced by actors depend on the order in which actors are fired. Note that as seen in some previous examples, allowing non-blocking reads is, however, not a sufficient criterion for nondeterminism. (In contrast, only allowing blocking reads is a sufficient criterion for deterministic actors.) In the following, DDF is used to refer to the class of nondeterministic actors.

In order to implement DDF actors, guard functions and multiple transitions with different token consumption and production rates can be used. In particular, it is possible to implement a nondeterministic merge actor. Note that in [Buc93], it is shown that BDF is already Turing complete. Concerning the analysis, every actor which has not been classified as CSDF, BDF, or deterministic, is assumed to be a DDF actor.

4.10 Related Work and Limitations

Models of Computation are an important concept in the design of embedded systems [LS98]. MoCs permit the use of efficient domain-specific optimization methods [LBS+11]. The advantages have been shown by many examples, e.g., for real time reactive systems [BFM+05] and in the signal processing domain [BB00b; BB00a]. An environment for modeling and simulating different and heterogeneous MoCs is provided by Ptolemy II [Pto14]. Actors are classified by the domain they are assigned to, and the domain is explicitly stated by a *director* that is responsible for proper actor invocation. In Ptolemy II, heterogeneous MoCs can be composed hierarchically, i.e. an actor can be refined by a network of actors, which is again controlled by a domain-specific director. Other approaches to model well-defined MoCs are also library-based and do not require actor classification, e.g., YAPI [dKES+00] and SHIM [ET05].

Other heterogeneous MoCs have been proposed. FunState [TSZ+99; STG+01], which has already been reviewed in Section 3.4, is of particular interest as it is similar to the proposed dataflow model. However, the authors only provide some kind of modeling guidelines to translate static and dynamic dataflow models as well as FSMs into FunState. The reverse, i.e., the actor classification, is neglected.

On the other hand, SystemC [GLMS02] is becoming the de facto standard for the design of digital hardware/software systems. While SystemC permits the modeling of many different MoCs, there is no unique representation of a particular MoC in SystemC. An approach for representing MoCs in SystemC is described in [HSV04]. They have implemented a custom library of channel types like rendezvous channels on top of the SystemC DE simulation kernel. However, implementing these channels on top of the SystemC DE simulation kernel

curtails the simulation performance. In order to improve the simulation efficiency, Patel et al. [PS04] extend SystemC itself with different simulation kernels for *Communicating Sequential Processes* [Hoa85] and *Finite State Machine MoCs*. Thus, the classification of actors is again based on explicit modeling primitives. In contrast, the proposed approach can be applied to actors derived from existing well-formed SystemC modules as described in Section 2.3.

One major problem, even when models are restricted to a subset of SystemC, is the unstructured use of communication primitives which makes the automatic classification of a SystemC design a hard problem. The SystemC transaction level modeling (TLM) standard [Acc12] does not alleviate these problems because it is not concerned with defining representations of MoCs in SystemC. Instead, the TLM-2.0 standard defines transaction level interfaces via method calls, therefore improving simulation efficiency and providing the foundation for platform-based design in SystemC. In this context, the work of Habibi et al. [HTS+06; HMT06] and Niemann and Haubelt [NHUT07] goes one step further. They use the TLM standard in combination with (Abstract) Finite State Machines which specify the behavior of single SystemC modules. In both cases, the focus is on the formalization of the entire SystemC transaction level model, and not on the classification of a single actor.

There are few publications on the classification of dataflow actors into known models of computation. This is not surprising, as this problem in its general form is not decidable. However, some approaches have been published since the proposed approach has been presented in [ZFHT08].

In [CPB12], an instrumentation-driven classification approach for DFGs is presented. To this end, dataflow models are instrumented such that traces of the communication patterns are recorded while executing the model. Subsequently, these communication traces are analyzed whether they correspond to the CSDF model of computation or not. Due to the incompleteness of simulation, this trace-based approach represents only a necessary condition for a given actor being a CSDF actor. In other words, depending on the input data, an actor may no longer adhere to the CSDF model of computation. In this case, treating the actor like a CSDF actor could introduce deadlocks into the model. This problem is avoided by the proposed classification algorithm, which statically analyzes the actor FSM and therefore represents a sufficient condition for an actor to adhere to a static dataflow model. Moreover, the proposed approach permits the static communication pattern to span multiple transitions, which is not considered by the trace-based approach. However, the trace-based approach does not depend on a certain representation of the underlying dataflow actor, whereas the proposed approach obviously requires a representation based on transitions with a static communication behavior.

The classification methodology presented in [WR10; WR12] is based on the CAL actor language reviewed in Section 3.4. Both papers improve the proposed

approach by performing an *abstract interpretation* of the CAL actor under consideration. Thus, more actors may be identified as SDF or CSDF actors compared to the proposed approach in principle, which only analyzes the static communication behavior of transitions. The basic classification algorithm, however, is similar to the proposed approach. In particular, SDF actors are constrained to actions that have the same input and output patterns. In contrast, the proposed approach permits the static communication pattern to span multiple transitions. In [WR10; WR12], it is also described how *quasi-static* actors can be detected. Quasi-static actors in this context are comparable to CSDF actors, with the key difference being that phases are not traversed cyclically. Instead, an arbitrary successor phase can be selected after the current phase is finished. In principle, a similar detection mechanism could be realized for the proposed dataflow model by allowing alternative communication patterns, as has been outlined for the SELECT and SWITCH actors of the BDF model in Section 4.4.2. However, as this quasi-static schedule (QSS) model is not a well-defined dataflow MoC, we abstain from discussing the analysis in more detail. In contrast, we mainly use QSSs for the representation of scheduling decisions as discussed in Section 5.3.

Finally, [SGE+13] describes how FSM-based SADF graphs can be extracted from disciplined dataflow networks (DDNs). Basically, DDNs restrict actor variables and input tokens to integer type, and require that they can only assume a finite number of values. The approach first identifies all possible scenarios of a DDN, and subsequently extracts their SDF graphs. Then, the possible sequences of executions of these scenarios are derived by means of a state-space exploration. Finally, an FSM is constructed for the identified scenario sequences. The classification methodology is applied to CAL actor networks. The analysis is based on firing rules as described in Section 4.8. In particular, the token value abstraction used in Section 4.8.2 is supported. Thus, this classification approach could be applied to the proposed dataflow model in principle.

5 System Synthesis

After having introduced the non-hierarchical dataflow model and some related analysis techniques, this chapter focuses on the hierarchical dataflow model which is used to incorporate design decisions into the model. Considering again the exemplary design flow supported by the proposed dataflow model as shown in Figure 5.1, Chapter 3 describes how an application can be modeled by means of the non-hierarchical dataflow model ("Application dataflow graph (DFG)"). Applicable to both, the hierarchical and non-hierarchical dataflow model, Chapter 4 outlines the representation and identification of less expressive dataflow models in order to support the decision-making process at system level. Note that the decision-making process itself as part of design space exploration is not discussed in the context of this thesis. This chapter therefore is mainly concerned with the refinement step at system level, i.e., the incorporation of binding and scheduling decisions into the dataflow model.

Binding decisions can be incorporated into the model by means of hierarchical DFGs ("Partitioned DFG"). To this end, the hierarchical actor model is introduced in Section 5.1. This step represents a structural refinement only, and actors are still executed in an unconstrained manner. The incorporation of binding decisions into the model is described in Section 5.2. Scheduling decisions can be

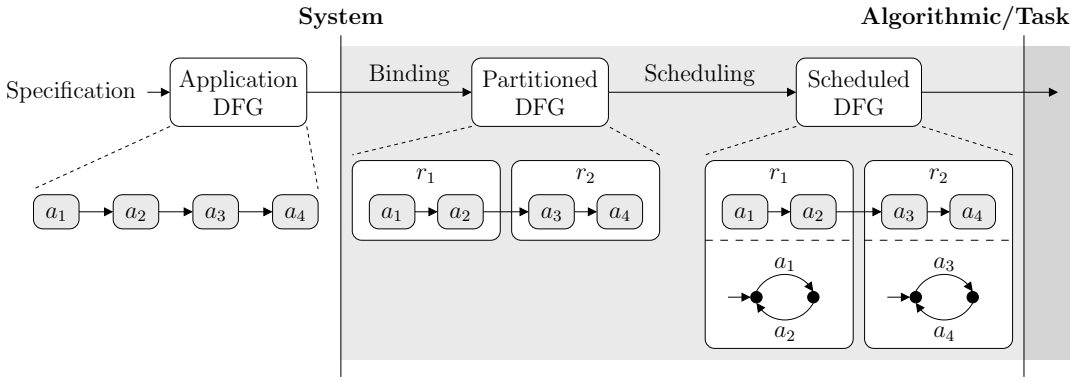


Figure 5.1: Design flow supported by the proposed model. Only the steps pertaining to system synthesis are shown (cf. Figure 2.2 on page 9).

incorporated into the model also by means of hierarchical DFGs ("Scheduled DFG"). This refinement step is described in Section 5.3 for various scheduling schemes. Results are presented in Section 5.4. Finally, Section 5.5 discusses related work and the limitations of the hierarchical modeling approach.

5.1 Hierarchical Model

In this section, the hierarchical actor model as introduced in [ZHF+13; ZHFT13] is described. At a glance, *composite* (i.e., hierarchical) actors consist of a structural part (ports, actors and channels), and a behavioral part (transitions). More formally, composite actors are defined as follows:

Definition 5.1 (Composite Actor)**.** A composite actor $a = (I, O, A, C, \mathrm{K}, \mathrm{D}, B, M, m_{\mathrm{cur}}, m_0, v, v_0, F_g, F_a, T, \mathrm{peek}, \mathrm{cons}, \mathrm{prod})$ extends the definition of leaf actors (cf. Definition 3.1) by a set of child actors A, a set of channels $C \subseteq A.O \times A.I$ connecting output ports of child actors with input ports of child actors[6], a function $\mathrm{K}\colon C \to \mathbb{N}$ which specifies the capacity of each channel, a function $\mathrm{D}\colon C \to \mathbb{N}_0$ which specifies the number of initial tokens on each channel, and a set of port-to-port bindings $B \subseteq (I \times A.I) \cup (A.O \times O)$ connecting input ports of the composite actor with input ports of child actors, and output ports of child actors with output ports of the composite actor, respectively.

Each input port of each child actor must be bound to exactly one channel or input port of the composite actor, and each output port of the composite actor must be bound to exactly one output port of a child actor:

$$\forall p \in A.I \cup O : |\{p' \in A.O \cup I \mid (p', p) \in C \cup B\}| = 1 \tag{5.1}$$

Analogously, each output port of each child actor must be bound to exactly one channel or output port of the composite actor, and each input port of the composite actor must be bound to exactly one input port of a child actor:

$$\forall p \in A.O \cup I : |\{p' \in A.I \cup O \mid (p, p') \in C \cup B\}| = 1 \tag{5.2}$$

As Conditions (5.1) and (5.2) induce a surjective mapping between a child actor port $p \in A.I \cup A.O$ and the channel $c \in C$ to which p is connected to, we use $\mathrm{D}(c)$ and $\mathrm{K}(c)$ interchangeably with $\mathrm{D}(p)$ and $\mathrm{K}(p)$ in the following. If p is bound to a port of the composite actor, this notation is implicitly extended to the parent actor of the composite actor, etc.

[6] $A.O$ denotes the set of all output ports of all child actors, i.e., $A.O = \bigcup_{a \in A} a.O$. This notation is also used for other sets, e.g., for input ports $A.I$ and transitions $A.T$.

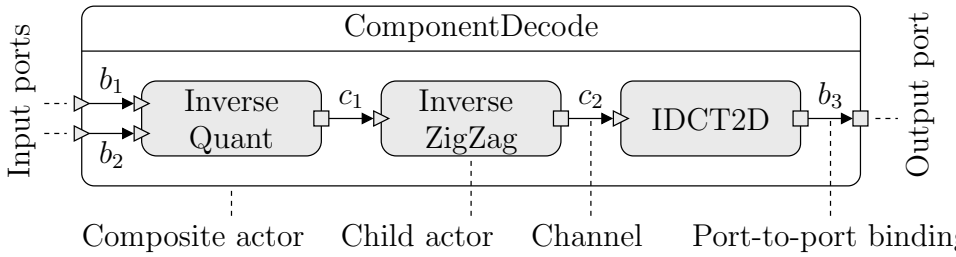


Figure 5.2: Composite actor which bundles the per-component decoding functionality of the JPEG decoder from Figure 2.3 on page 12.

Example 5.1. Figure 5.2 shows a composite actor `ComponentDecode` which contains those actors from Figure 2.3 on page 12 that are responsible for decoding the image components (Y, Cb, Cr) of a JPEG image. By instantiating the resulting composite actor more than once, the parallel processing of image components can be modeled (and synthesized). This particular composite actor consists of the child actors `InverseQuant`, `InverseZigZag`, and `IDCT2D`. The ports of the child actor are connected by channels c_1 and c_2, and to the ports of the composite actor by port-to-port bindings b_1, b_2, and b_3. Note that the `IDCT2D` actor could be in turn hierarchically refined as shown in Figure 4.11 on page 75.

In the following, a composite actor without transitions (i.e., $T = \emptyset$) is called *structural composite actor*, whereas a composite actor with $T \neq \emptyset$ will be referred to as *functional composite actor*. While the set of leaf actors models the functionality of the application (cf. actors a_1–a_4 in Figure 5.1), structural composite actors can be used to model binding decisions (cf. actors r_1 and r_2 in Figure 5.1), and functional composite actors are used to represent scheduling decisions (cf. finite state machines (FSMs) in Figure 5.1). Note that a leaf actor is, in fact, a composite actor without child actors (i.e, $A = \emptyset$), and therefore, $C = \emptyset$ and $B = \emptyset$.

Example 5.2. The composite actor `ComponentDecode` shown in Figure 5.2 is a structural composite actor as indicated by the absence of an actor FSM. In turn, the child actors `InverseQuant`, `InverseZigZag`, and `IDCT2D` are not scheduled (yet).

As will be seen, a functional composite actor is only allowed to schedule its immediate child actors. Thus, structural composite actors cannot be child actors of functional composite actors in the proposed dataflow model, as structural composite actors do not have transitions, and therefore cannot be scheduled by a

hypothetical functional composite parent actor. Thus, if a structural composite child actor needs to be scheduled, it must be either (recursively) dissolved by inlining its child actors into the functional composite parent actor, or a scheduling scheme must be (recursively) annotated to the structural composite child actor.

The parent-child relation of actors induces a directed graph $G_A = (V_A, E_A)$, where the vertices V_A correspond to the actors of an application, and for each composite actor a and each child actor $a' \in a.A$, an edge (a, a') is added to E_A. We require that the induced graph does not contain cycles and that each child actor has exactly one parent actor, i.e., G_A must be a tree. In the following, we assume the existence of a *root actor* a_{root}, which is defined as a composite actor with no ports, i.e, $I = O = \emptyset$. The root actor represents the overall application. In turn, this means that each port of an actor is eventually bound to a channel of a composite actor. In the following, we write $a \preceq a'$ if a is an ancestor of a' w.r.t. G_A. Note that $a \preceq a'$ also if $a = a'$. Thus, "$\preceq$" defines a (non-strict) partial order over the the set of actors comprising the application. The root actor a_{root} is an ancestor of every nested actor a (including a_{root}).

5.1.1 Operational Semantics

Transitions are kept unchanged for composite actors compared to leaf actors (cf. Definition 3.5). In particular, this means that transitions of a functional composite actor also have a static communication behavior. In turn, functional composite actors can be treated as leaf actors for analytic purposes. In order to support the scheduling of child actors, guard functions and action functions are extended for functional composite actors by the concept of *tasks* and *task lists*. Basically, tasks represent transitions of child actors, while task lists specify dependencies between tasks. Tasks and task lists are created by functional composite actors, and can be used in guard functions and action functions to implement a wide range of scheduling schemes, as will be seen in Section 5.3.

Definition 5.2 (Task)**.** Given a functional composite actor a. Then, a task τ of a is a non-empty set of transitions of a child actor $a' \in a.A$, i.e., $\tau \subseteq a'.T$.

In order to simplify the notation, we write $\tau = a'$ in the special case that $\tau = a'.T$. In other words, specifying an actor a' as task is equivalent to specifying all of its transitions $a'.T$. In the following, we use $\text{parent}(\tau) = a'$ to refer to the child actor $a' \in a.A$ such that $\tau \subseteq a'.T$.

Definition 5.3 (Task List)**.** Given a functional composite actor a. Then, a task list $\lambda = (\Gamma, \prec)$ of a is a strict partial order "$\prec$" over a multiset of tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of a. Each task $\tau \in \Gamma$ is a non-empty set of transitions of a child actor $a' \in a.A$ according to Definition 5.2.

Algorithm 5.1 Evaluation of a task τ

```
1: procedure EVALUATE(Task τ)
2:     for all t ∈ τ do
3:         if EVALUATE(t) = ⊤ then
4:             Remember t as enabled transition
5:             return ⊤
6:         end if
7:     end for
8:     return ⊥
9: end procedure
```

Algorithm 5.2 Evaluation of a task list λ

```
1: procedure EVALUATE(Task list λ)
2:     for all τ ∈ Γ_eval(λ) do
3:         if EVALUATE(τ) = ⊥ then
4:             return ⊥
5:         end if
6:     end for
7:     return ⊤
8: end procedure
```

A task list represents dependencies between tasks: If for any two tasks τ_i and τ_j, $\tau_i \prec \tau_j$, task τ_i must be executed and finished before task τ_j can be evaluated or executed. In the following, $\Gamma_{\text{pred}}(\lambda = (\Gamma, \prec), \tau) = \{\tau' \in \Gamma \mid \tau' \prec \tau\}$ denotes the set of *predecessor tasks* of a task τ w.r.t. a task list λ. Note that Γ is defined as a multiset in order to accommodate multiple occurrences of the same transition set.

Evaluation phase

The evaluation phase of a task τ is summarized by Algorithm 5.1: Each transition $t \in \tau$ is evaluated as described in Section 3.1.1. If at least one transition $t \in \tau$ is enabled, task τ is enabled. In contrast, if no transition $t \in \tau$ is enabled, task τ is not enabled.

Note that if an enabled transition t is found during the evaluation of a task τ, this transition may be cached for the subsequent execution phase. However, after the execution phase, τ must be re-evaluated, because the cached transition may be no longer enabled in the general case. However, if we consider only conflict-free DFGs, this situation can only arise if a (possibly different) task τ' is executed such that $\text{parent}(\tau) = \text{parent}(\tau')$.

The evaluation phase of a task list λ is summarized by Algorithm 5.2: Basically, all tasks without predecessor tasks are evaluated. Note that only tasks without predecessor tasks are evaluated, because evaluating tasks with predecessor tasks

Algorithm 5.3 Execution of a task τ

```
1: procedure EXECUTE(Task τ)
2:     Let t be an enabled transition of τ
3:     EXECUTE(t)
4: end procedure
```

Algorithm 5.4 Execution of a task list λ

```
1: procedure EXECUTE(Task list λ)
2:     Let ⟨τ_1, ..., τ_n⟩ ← topSort(λ)
3:     for i ← 1, n do
4:         if Γ_pred(λ, τ_i) ≠ ∅ then
5:             EVALUATE(τ_i)
6:         end if
7:         EXECUTE(τ_i)
8:     end for
9: end procedure
```

would obviously require *executing* the predecessor tasks, which is not allowed during the evaluation phase. Thus, the set of tasks to evaluate is:

$$\Gamma_{\text{eval}}(\lambda = (\Gamma, \prec)) = \{\tau \in \Gamma \mid \Gamma_{\text{pred}}(\lambda, \tau) = \emptyset\}$$

A task list λ is enabled if all tasks $\tau \in \Gamma_{\text{eval}}(\lambda)$ are enabled. Otherwise, λ is not enabled.

As the evaluation of transitions is side-effect free according to the semantics of the proposed dataflow model, the evaluation of tasks and task lists is also side-effect free w.r.t. the state of the dataflow model. Thus, transitions of tasks, as well as tasks of task lists can be evaluated in parallel in principle.

Execution phase

The execution phase of an enabled task τ is summarized by Algorithm 5.3. Basically, the enabled transition found during the evaluation phase is executed as described in Section 3.1.1.

Finally, the execution phase of an enabled task list λ is summarized by Algorithm 5.4. Basically, all tasks in Γ are executed. Remember that tasks with predecessor tasks must still be evaluated. However, during the execution phase, this poses no problem, and a task is evaluated after all predecessor tasks have been executed. In contrast to the evaluation phase, we now expect that evaluation always succeeds, i.e., each task must be enabled. Otherwise, the model behavior is undefined. Let $\text{topSort}(\lambda = (\Gamma, \prec)) = \langle \tau_1, \ldots, \tau_n \rangle$ denote a topological ordering of the task set Γ w.r.t. the strict partial order "$\prec$". Note that such a

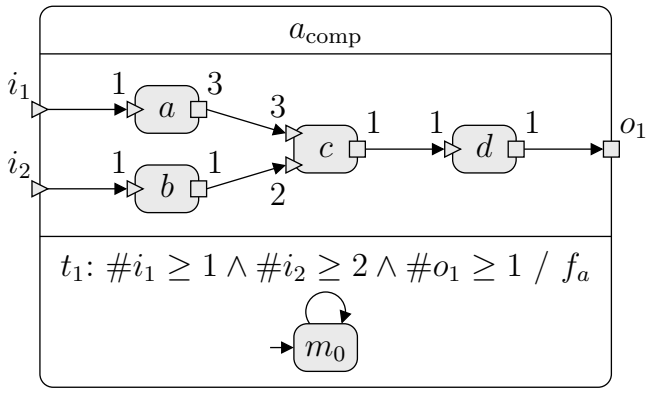


Figure 5.3: Example functional composite actor a_{comp} with SDF child actors a–d.

topological ordering can always be found for a strict partial order, as the induced dependency graph is acyclic. Then, τ_1 is executed (it must have been evaluated during the evaluation phase due to having no predecessor tasks). Subsequently, τ_2 is evaluated (if $\Gamma_{\text{pred}}(\lambda, \tau_2) = \{\tau_1\}$) and executed, and so on. When the last task τ_n has been evaluated and executed, the execution of λ is finished.

Example 5.3. Consider the functional composite actor depicted in Figure 5.3. It consists of four synchronous dataflow (SDF) child actors with the annotated token consumption and production rates. Thus, a possible periodic static order schedule (PSOS) as described in Section 4.2 is $S = \langle a, b^2, c, d \rangle$ which fires actor a once, actor b twice, actor c once, and finally, actor d once. A possible task list $\lambda = (\Gamma, \prec)$ which implements S consists of $\Gamma = \{\tau_1 = a, \tau_2 = b, \tau_3 = b, \tau_4 = c, \tau_5 = d\}$, and $\tau_1 \prec \tau_2 \prec \tau_3 \prec \tau_4 \prec \tau_5$. This means that a must be executed first, followed by the first firing of b, and so on. It should be noted that a PSOS represents a total ordering on the actor firings. In turn, "$\prec$" also induces a total ordering on tasks in a task list when implementing a PSOS. Finally, the task list λ could be evaluated and executed by the action function f_a of the (only) transition t_1 of the given composite actor. In this case, evaluating λ results in evaluating task τ_1, which is the only task without predecessor tasks. Subsequently, when executing λ, τ_1 is executed first (it is already evaluated), followed by the evaluation and execution of τ_2, etc. When τ_5 has been executed, the execution of λ is finished.

Forwarding of Token Sequences

Until now, the forwarding of tokens from/to ports of the composite actor to/from ports of child actors, as well as the forwarding of tokens from output ports of

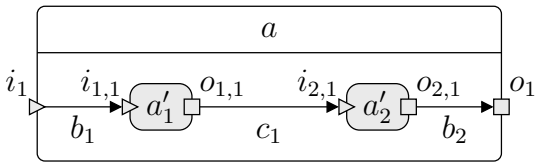


Figure 5.4: Consumption and production of tokens in a hierarchical model.

child actors to input ports of child actors has been neglected. While the former is represented by port-to-port bindings, the latter is represented by channels (cf. Definition 5.1).

To this end, we assume that a composite actor a and two child actors $a'_1, a'_2 \in a.A$ are given as shown in Figure 5.4. Concerning the channel c_1, token sequences produced by actor a'_1 on $o_{1,1}$ are simply forwarded to input port $i_{2,1}$ of actor a'_2 by means of a first in, first out (FIFO) buffer associated with channel c_1. This behavior is basically the same for structural composite actors and functional composite actors.

Concerning the port-to-port bindings b_1 and b_2, two cases can be distinguished: If a is a structural composite actor, token sequences are simply forwarded from input port i_1 of the composite actor a to input port $i_{1,1}$ of the child actor a'_1. Analogously, token sequences are simply forwarded from output port $o_{2,1}$ of the child actor a'_2 to output port o_1 of the composite actor a.

If a is a functional composite actor, the token sequences provided to the guard function f_g and action function f_a of a transition t of the composite actor are based on the values of $\text{peek}(p, f_g)$ and $\text{cons}(p, f_a)$ for a given input port $p \in I$ as described in Section 3.1.1 (cf. Algorithm 3.1 and 3.2). Basically, f_g and f_a are free to transform these token sequences before forwarding them to input ports of child actors that are bound to input ports of the composite actor. Transitions of child actors are then evaluated based on these (possibly transformed) token sequences. Note that the action function f_a is also responsible for producing a token sequence based on the value of $\text{prod}(p, f_a)$ for a given output port p of the composite actor (cf. Algorithm 3.2). Again, f_a is free to transform token sequences from output ports of child actors that are bound to output ports of the composite actor in order to accomplish this task.

In principle, an implementation is free to provide appropriate token forwarding mechanisms. For example, an existing implementation allows FIFO buffers to be associated with a port-to-port binding, analogously to channels. In addition, as discussed in Section 6.2, this implementation allows guard functions and action functions to randomly access the tokens on these FIFO buffers in the range specified by the values of peek, cons, and prod, respectively. Thus, an action function can read a token in a FIFO channel (via an actor input port), transform the token, and place it in the FIFO buffer associated with a port-to-

port binding, where it can be subsequently consumed by transitions of a child actor. Analogously, an action function can read a token in the FIFO buffer associated with a port-to-port binding, transform it, and subsequently store it in a FIFO channel (via an actor output port). However, when using such an implementation-defined forwarding mechanism, the designer has to ensure that Requirement 3.1 is satisfied. Otherwise, the model behavior may be undefined.

Remember that transitions of functional composite actors have the same semantics as transitions of leaf actors (cf. Definition 3.5). From an analytical point of view, a concrete token forwarding mechanism therefore does not influence the static communication behavior of transitions. In particular, this means that the same analysis and optimization techniques can be applied to leaf actors and functional composite actors.

5.2 Binding

In order to describe binding decisions, we assume a set of *computing resources* $R = \{r_1, \ldots, r_n\}$ that are available in the target platform, like processor cores etc. Furthermore, we assume that the function $\beta\colon V_A \to R$ specifies for each actor $a \in V_A$ of a hierarchical dataflow model the resource $\beta(a) \in R$ to which a is bound to. The values of β can be determined, e.g., by an automatic design space exploration, or specified by the user. In particular, child actors of a composite actor may be bound to different resources.

Although structural composite actors do not have any functionality, they may be bound to a resource. In this way, structural composite actors can be conveniently used to represent binding decisions, as child actors of a composite actor which are not bound to a resource *inherit* the binding of their parent composite actor. In order to indicate that an actor is not bound to a resource, a special resource r_{none} is used in the following. In order to simplify the notation, it is assumed in the following that the values of β correspond to a *normalized* representation where all functional composite actors (and leaf actors) are bound to a resource $r \neq r_{\text{none}}$, while all structural composite actors are not bound to a resource.

The actor hierarchy graph $G_A = (V_A, E_A)$ and the binding of actors to resources induces a directed resource dependency graph $G_R = (V_R, E_R)$, where the vertices V_R correspond to the resources R, and for each edge $(a, a') \in E_A$, an edge $(\beta(a), \beta(a'))$ is added to E_R if $\beta(a) \neq \beta(a')$. Note that the latter condition prevents self-loops in G_R. These are easily handled as will be shown in the following, and can therefore be omitted from G_R. Moreover, r_{none} is either the root vertex of G_R, or does not appear in G_R at all, depending on whether the root actor a_{root} is a structural composite actor or not.

In order to derive simple operational semantics and an efficient runtime environment (RTE), we assume in the following that any vertex $r \in V_R$ of the

resource dependency graph G_R does have at most one incoming edge: If a vertex $r \in V_R$ has multiple incoming edges $(r_1, r) \in E_R$ and $(r_2, r) \in E_R$ such that $r_1 \neq r_2$, resources r_1 and r_2 may compete for the same resource r. In this case, an additional scheduling mechanism would be required for resource r which arbitrates requests from resources r_1 and r_2. While possible in principle, the operational semantics would become more complex. In the following, the necessary changes are outlined in order to support such a scenario.

5.2.1 Operational Semantics

As the transitions of a task τ belong to the same actor, the operational semantics of tasks are not refined compared to Section 5.1.1. In other words, the transitions of τ are evaluated on resource $\beta(\text{parent}(\tau))$.

On the other hand, the tasks in a task list may have different parent actors. Thus, depending on the binding of actors to resources, the evaluation and execution of tasks in a task list must be delegated to the appropriate resources.

Evaluation Phase

Algorithm 5.5 summarizes the binding-aware evaluation phase of a task list τ when executed by an actor bound to a resource r_{self}. To this end, $\Gamma_{\text{eval}}(\lambda, r) = \{\tau \in \Gamma_{\text{eval}}(\lambda) \mid \beta(\tau) = r\}$ denotes the set of tasks to be evaluated on a given resource r. Remember that $\Gamma_{\text{eval}}(\lambda)$ refers to the tasks of λ without predecessor tasks. First, the evaluation of tasks which are bound to different resources is started asynchronously on the corresponding resources (cf. lines 2–6). Subsequently, tasks bound to r_{self} are evaluated synchronously on r_{self} (cf. lines 7–13). Finally, we wait for the completion of the asynchronously evaluated tasks (cf. lines 14–21). If the overall result res is already $\perp$ after the partial evaluation of tasks, the evaluation of the remaining tasks can be skipped or canceled in principle. However, this behavior depends on the implementation (and target resources), and is not reflected by Algorithm 5.5.

In order to support multiple incoming edges in the resource dependency graph G_R as discussed above, lines 14–21 could no longer just wait until all pending tasks are finished that are evaluated asynchronously on different resources. Instead, task evaluation or execution requests from other resources that are posted to r_{self} would have to be processed here in order to prevent the introduction of deadlocks into the model (in the case of cyclic resource dependencies).

Execution Phase

Algorithm 5.6 summarizes the binding-aware execution phase of a task list τ when executed by an actor bound to a resource r_{self}. Analogously to $\Gamma_{\text{eval}}(\lambda, r)$,

Algorithm 5.5 Binding-aware evaluation of a task list λ on a resource r

```
1: procedure EVALUATE(Task list λ = (Γ, ≺), Resource r_self)
2:     for all r ∈ R do                          ▷ Asynchronous task evaluation
3:         if r ≠ r_self ∧ Γ_eval(λ, r) ≠ ∅ then
4:             Asynchronously start the evaluation of tasks Γ_eval(λ, r) on resource r
5:         end if
6:     end for
7:     Let res ← ⊤                               ▷ Synchronous task evaluation
8:     for all τ ∈ Γ_eval(λ, r_self) do
9:         if EVALUATE(τ) = ⊥ then
10:            res ← ⊥
11:            break
12:        end if
13:    end for
14:    for all r ∈ R do
15:        if r ≠ r_self ∧ Γ_eval(λ, r) ≠ ∅ then
16:            Wait for completion of tasks Γ_eval(λ, r) on resource r with result res_r
17:            if res_r = ⊥ then
18:                res ← ⊥
19:            end if
20:        end if
21:    end for
22:    return res
23: end procedure
```

we use $\Gamma_{\text{exec}}(\lambda = (\Gamma, \prec), r) = \{\tau \in \Gamma \mid \beta(\tau) = r\}$ to denote the set of tasks to be executed on a given resource r. First, the execution of tasks which are bound to different resources is started asynchronously on the corresponding resources (cf. lines 2–6). Subsequently, tasks bound to r_{self} are executed synchronously on r_{self} (cf. lines 7–17). Note that in contrast to Algorithm 5.4, we now have to wait for the completion of predecessor tasks that are executed on resources other than r_{self} (cf. lines 11–13). Finally, we wait for the completion of the asynchronously executed tasks (cf. lines 18–22).

Again, in order to support multiple incoming edges in the resource dependency graph G_R as discussed above, both places where we wait for pending tasks to complete on other resources (i.e., lines 11–13 and lines 18–22) would have to process task evaluation or execution requests from other resources that are posted to r_{self} in order to prevent the introduction of deadlocks into the model (in the case of cyclic resource dependencies).

Example 5.4. Consider again the functional composite actor depicted in Figure 5.3. A slightly different static scheduling scheme compared to the PSOS

Algorithm 5.6 Binding-aware execution of a task list λ on a resource r

```
1: procedure EXECUTE(Task list λ = (Γ, ≺), Resource r_self)
2:     for all r ∈ R do                          ▷ Asynchronous task execution
3:         if r ≠ r_self ∧ Γ_exec(λ, r) ≠ ∅ then
4:             Asynchronously start the execution of tasks Γ_exec(λ, r) on resource r
5:         end if
6:     end for
7:     Let Γ_self ← Γ_exec(λ, r_self)            ▷ Synchronous task execution
8:     Let ⟨τ_1, ..., τ_n⟩ ← topSort(Γ_self, ≺)
9:     for i ← 1, n do
10:        if Γ_pred(λ, τ_i) ≠ ∅ then
11:            for all τ_pred ∈ Γ_pred(λ, τ_i) \ Γ_self do
12:                Wait for completion of task τ_pred on resource β(parent(τ_pred))
13:            end for
14:            EVALUATE(τ_i)
15:        end if
16:        EXECUTE(τ_i)
17:    end for
18:    for all r ∈ R do
19:        if r ≠ r_self ∧ Γ_exec(λ, r) ≠ ∅ then
20:            Wait for completion of tasks Γ_exec(λ, r) on resource r
21:        end if
22:    end for
23: end procedure
```

$S = \langle a, b^2, c, d \rangle$ (cf. Example 5.3) consists of the task list $\lambda = (\Gamma, \prec)$ with $\Gamma = \{\tau_1 = a, \tau_2 = b, \tau_3 = b, \tau_4 = c, \tau_5 = d\}$, $\tau_1 \prec \tau_4 \prec \tau_5$, and $\tau_2 \prec \tau_3 \prec \tau_4$. In this case, the partial order "$\prec$" is described by means of two *chains*. A *chain* is a totally ordered subset of a partially ordered set. In contrast to the total order induced by the PSOS S, the two firings of b do not depend on the firing of a in this case. Thus, in principle, they can be performed in parallel, depending on the binding of actors to resources. Moreover, both tasks τ_1 and τ_2 don't have any predecessor tasks in this case.

In order to illustrate the binding-aware execution of λ, assume a set of resources $R = \{r_1, r_2\}$, and the following values of β: $\beta(a_{\text{comp}}) = \beta(a) = \beta(c) = \beta(d) = r_1$, and $\beta(b) = r_2$, i.e., all actors except b are bound to resource r_1, while b is bound to resource r_2. Then, the sequence diagram in Figure 5.5 shows the possible execution of λ during the execution of the action function f_a of transition t_1 of actor a_{comp} (cf. Figure 5.3). First, the evaluation of $\tau_2 = b$ is started on resource r_2, followed by the evaluation of $\tau_1 = a$ on resource r_1. Assuming that the evaluation succeeds, τ_2 and τ_3 are executed on r_2, while τ_1,

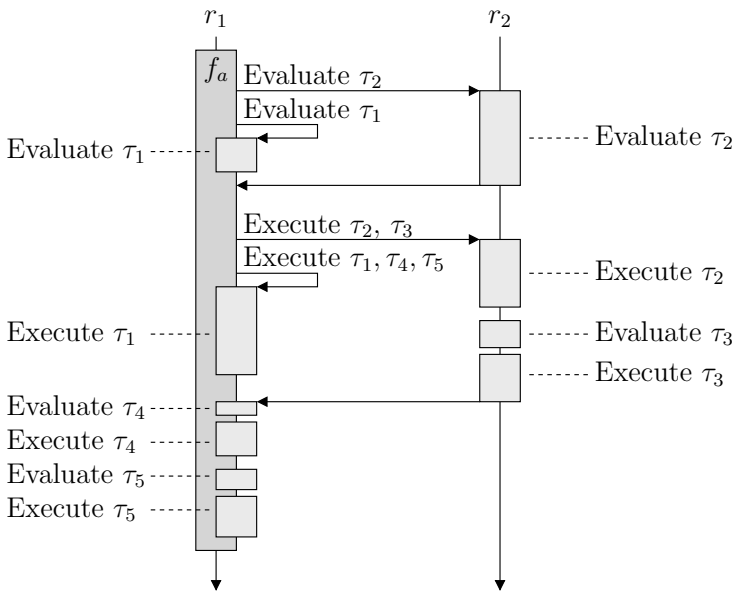


Figure 5.5: Binding-aware execution of the action function f_a from Figure 5.3

τ_4 and τ_5 are executed on r_1. Note that the evaluation and execution of τ_4 is started only after τ_3 is finished on r_2.

5.2.2 Runtime Environment

Depending on the resource dependency graph G_R, two types of resources can be distinguished: *Initiator resources* are resources where at least one actor is bound to that is not scheduled by its parent actor. Given the directed graph of actors $G_A = (V_A, E_A)$ representing the actor hierarchy, this set of actors is defined for a given resource r as follows:

$$A_{\text{initiator}}(r) = \{a \in G_A \mid \beta(a) = r \wedge (\nexists a' \in G_A : (a', a) \in E_A \wedge \beta(a') \neq r_{\text{none}})\}$$

Thus, a resource r is an initiator resource if $A_{\text{initiator}}(r) \neq \emptyset$. In this case, the RTE of r consists of an implementation-defined scheduling scheme for the actors in $A_{\text{initiator}}(r)$. As implemented, a simple round-robin dynamic scheduling scheme is used which is summarized by Algorithm 5.7.

On the other hand, *target resources* are resources where at least one actor is bound to that is scheduled by its parent actor, and the parent actor is bound to a different resource. Given the directed graph of actors $G_A = (V_A, E_A)$ representing the actor hierarchy, this set of actors is defined for a given resource r as follows:

$$A_{\text{target}}(r) = \{a \in G_A \mid \beta(a) = r \wedge (\exists a' \in G_A : (a', a) \in E_A \wedge \beta(a') \neq r_{\text{none}} \wedge \beta(a') \neq r)\}$$

Algorithm 5.7 Runtime environment of an initiator resource

```
 1: procedure RteInitiator(Resource r_self)
 2:     loop
 3:         for all a ∈ A_initiator(r_self) do
 4:             Let τ ← a.T
 5:             if Evaluate(τ) = ⊤ then
 6:                 Execute(τ)
 7:             end if
 8:         end for
 9:     end loop
10: end procedure
```

Algorithm 5.8 Runtime environment of a target resource

```
 1: procedure RteTarget(Resource r_self)
 2:     loop
 3:         Wait for task evaluation or execution request
 4:         if Received task evaluation request Γ_eval(λ, r_self) then
 5:             Evaluate tasks Γ_eval(λ, r_self) according to lines 7–13 of Algorithm 5.5
 6:         else if Received task execution request Γ_exec(λ, r_self) then
 7:             Execute tasks Γ_exec(λ, r_self) according to lines 7–17 of Algorithm 5.6
 8:         end if
 9:     end loop
10: end procedure
```

Thus, a resource r is a target resource if $A_{\text{target}}(r) \neq \emptyset$. Note that there may exist actors bound to a resource r that are neither contained in $A_{\text{initiator}}(r)$ nor in $A_{\text{target}}(r)$. These are actors that are scheduled by actors bound to the same resource r, which is handled by Algorithms 5.5 and 5.6. A target resource processes task evaluation and execution requests for actors in $A_{\text{target}}(r)$ as summarized by Algorithm 5.8.

In order to support multiple incoming edges in the resource dependency graph, the RTEs would have to accommodate the following scenarios:

- Algorithm 5.8 may have to queue and process task evaluation and execution requests from different resources. This could be accomplished rather easily, e.g., by means of a first-come, first-served (FCFS) scheduling scheme.

- The more difficult scenario consists in a resource r for which $A_{\text{initiator}}(r) \neq \emptyset$ and $A_{\text{target}}(r) \neq \emptyset$. In this case, resource r is an initiator resource and a target resource at the same time, which requires the merging of Algorithm 5.7 and Algorithm 5.8. Basically, this requires the use of a suitable scheduling scheme which periodically executes the actors in $A_{\text{initiator}}(r)$,

but also processes aperiodic task evaluation and execution requests for actors in $A_{\text{target}}(r)$.

- Finally, cyclic resource dependencies are more difficult to handle, as task evaluation and execution requests may have to be processed while evaluating and executing a task. This problem has been addressed in the context of Algorithm 5.5 and Algorithm 5.6.

5.3 Scheduling

In [Lee89], the scheduling taxonomy shown in Table 5.1 has been introduced. In the general case, actors must be *assigned* to a specific computing resource. Subsequently, actors assigned to the same resource must be *ordered*. Finally, the *time* at which they fire must be determined.

Fully dynamic scheduling performs all tasks at run time. In particular, actors are bound to resources at run time. In Section 5.2, the binding-aware operational semantics of the proposed dataflow model have been described. However, a static binding has been assumed. In principle, it is possible to extend the approach to also provide for a dynamic binding of actors to resources. This is mainly a task of the underlying RTE. Here, numerous approaches have been proposed [LKP+10; HN12; HT12; SGB+12; HHB+12; QP13]. In particular, actors can be easily remapped after a transition has been executed, as in this case, only the actor state has to be transferred, and no local variables which may be used during the execution of a guard function or an action function. In fact, a related approach is described in Chapter 6.3.4.

Static-assignment scheduling performs the binding of tasks at compile time, while the ordering and timing of actor firings is performed at run time. This scheduling class has also been referred to as *dynamic scheduling* in previous sections. Note that this is the default scheduling scheme used for initiator resources as described in Section 5.2.2.

In contrast, *self-timed* scheduling determines the ordering of actors at compile time. This corresponds to the periodic static order scheduling as described in

Scheduling class	Scheduling task		
	Assignment	Ordering	Timing
Fully Dynamic	Run	Run	Run
Static-assignment	Compile	Run	Run
Self-timed	Compile	Compile	Run
Fully static	Compile	Compile	Compile

Table 5.1: Scheduling taxonomy according to [Lee89].

Section 4.2. Note that the timing of actor firings is still deferred to run time. For this class of schedules, Example 5.3 already illustrated how such schedules can be represented in the proposed dataflow model.

An additional class of schedules, namely quasi-static schedules (QSSs) are not reflected by Table 5.1. A QSS defers data-dependent scheduling decisions to run time, while data-independent scheduling decisions are performed at compile time. Thus, QSSs reside between static-assignment schedules and self-timed schedules according to Table 5.1. As will be seen, transitions can be also used to represent such schedules in the proposed dataflow model.

Finally, *fully static* scheduling performs all tasks at compile time. In particular, the timing of actor firings is determined at compile time. This is currently not supported by the proposed approach, where actor firings are either functionally synchronized by means of tokens transmitted over channels, or by the use of task lists, where synchronization is based on a partial ordering between tasks. Note that this approach would require to compute exact (or worst-case) execution times of guard functions and action functions in addition to token access times, which may be difficult in case of data-dependent control flow.

5.3.1 Self-Timed Scheduling

In this section, two self-timed scheduling strategies are discussed, namely *periodic static order scheduling* and *periodic partial order scheduling*. While the former corresponds to a total ordering of actor firings, the latter corresponds to a strict partial ordering of actor firings.

Periodic Static Order Scheduling

In order to implement a periodic static order schedule $S = \langle a_1, \ldots \rangle$ for a set of cyclo-static dataflow (CSDF) actors a_1–a_n, a functional composite actor a_{comp} can be used which consists of a single self-loop transition t_1 from the initial mode m_0 (cf. Figure 5.3). The guard function is set to $f_\top$, and the action function evaluates and executes a task list $\lambda = (\Gamma, \prec)$ which consists of a total ordering of all actor firings as specified by S. In the following, it is assumed that each task $\tau \in \Gamma$ only consists of a single transition t_k corresponding to phase k of the CSDF actor $a = \text{parent}(\tau) \in a_{\text{comp}}.A$. In particular, this is the case if a is a normalized CSDF actor as described in Section 4.3.2.

In order to determine the token production and consumption rates of the action function f_a associated with the only transition t_1 of the composite actor a_{comp}, only ports of the CSDF child actors that are bound to ports of a_{comp} according to the set of port-to-port bindings $a_{\text{comp}}.B$ have to be considered. To

this end, the token consumption and production rates of a task $\tau = \{t_k\}$ w.r.t. a port $p \in I \cup O$ of a_{comp} can be determined as follows:

$$\text{cons}(p, \tau = \{t_k\}) = \sum_{p' \in \text{parent}(\tau).I \colon (p,p') \in B} \text{cons}(p', t_k.f_a) \tag{5.3}$$

$$\text{prod}(p, \tau = \{t_k\}) = \sum_{p' \in \text{parent}(\tau).O \colon (p',p) \in B} \text{prod}(p', t_k.f_a) \tag{5.4}$$

Note that each sum contains at most one element according to Conditions (5.1) and (5.2). Then, the token consumption and production rates of f_a w.r.t. a port $p \in I \cup O$ of a_{comp} can be determined based on the task list $\lambda = (\Gamma, \prec)$ as follows:

$$\text{cons}(p, f_a) = \sum_{\tau \in \Gamma} \text{cons}(p, \tau) \tag{5.5}$$

$$\text{prod}(p, f_a) = \sum_{\tau \in \Gamma} \text{prod}(p, \tau) \tag{5.6}$$

Example 5.5. Consider the functional composite actor γ_1 depicted in Figure 5.6 which consists of the homogeneous synchronous dataflow (HSDF) actors a_1–a_4. A possible PSOS is $S_1 = \langle a_1, a_2, a_3, a_4 \rangle$. The task list $\lambda = (\Gamma, \prec)$ executed by the action function f_{a1} of transition t_1 then consists of $\Gamma = \{\tau_1 = a_1, \tau_2 = a_2, \tau_3 = a_3, \tau_4 = a_4\}$, and $\tau_1 \prec \tau_2 \prec \tau_3 \prec \tau_4$. According to Equation (5.5), it follows that $\text{cons}(i_1, f_{a1}) = \text{cons}(i_1, \tau_1) + \text{cons}(i_1, \tau_2) + \text{cons}(i_1, \tau_3) + \text{cons}(i_1, \tau_4) = 0 + 0 + 0 + 1 = 1$. Analogously, it follows that $\text{prod}(o_1, f_{a1}) = 1$.

Periodic Partial Order Scheduling

A PSOS is usually determined for each resource where multiple static actors are bound. Thus, in the presence of multiple resources, multiple PSOS are determined and implemented in a distributed manner. However, in the general case, this may introduce deadlocks into the model if the guarded actions semantics of the proposed dataflow model are imposed on the distributed PSOS.

Example 5.6. Consider again Figure 5.6. If γ_1 implements the PSOS $S_1 = \langle a_1, a_2, a_3, a_4 \rangle$ (cf. Example 5.5), it can be observed that a deadlock has already been introduced into the model, regardless of the schedule implemented by γ_2 (which may be a similar PSOS $S_2 = \langle a_5, a_6, a_7, a_8 \rangle$). This is due to the fact that in order to execute transition t_1 of γ_1, one token must be available on channel c_2. However, this token is produced by actor a_7, which can only fire after actor a_6 has been fired, which in turn can fire only if one token is available on channel

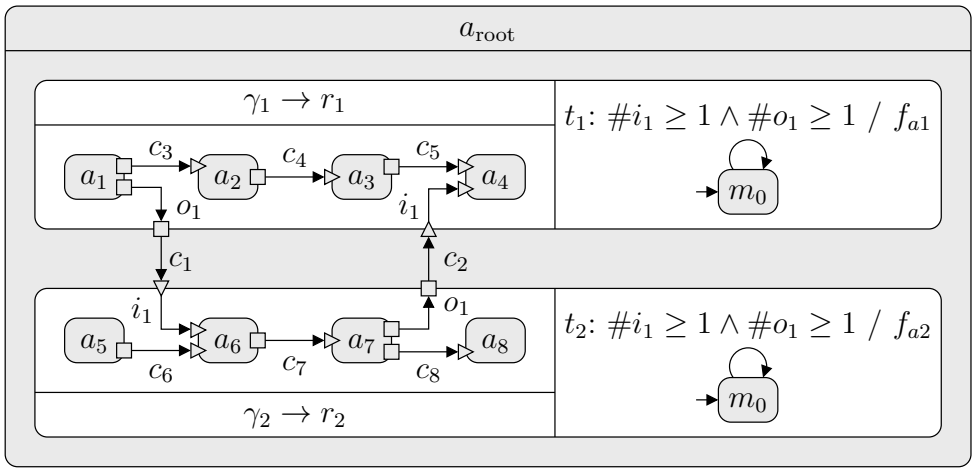


Figure 5.6: HSDF actors a_1–a_8 and functional composite actors γ_1 and γ_2. There are no initial tokens on any channel c_1–c_8. Actor γ_1 (and actors a_1–a_4) are bound to resource r_1, while actor γ_2 (and actors a_5–a_8) are bound to resource r_2.

c_1. However, this token is also produced by transition t_1 of γ_1, which cannot fire due to the missing token on channel c_2. As will be seen in Section 5.3.2, one possibility to solve this problem is to split S_1 into two partial PSOS, e.g., $S_{1,1} = \langle a_1, a_2, a_3 \rangle$ and $S_{1,2} = \langle a_4 \rangle$. Note that $S_{1,1}$ and $S_{1,2}$ then correspond to two transitions $t_{1,1}$ and $t_{1,2}$, respectively. Then, transition $t_{1,1}$ can be executed first, followed by transition t_2 of γ_2, followed by the execution of $t_{1,2}$.

The example shows that while it is possible to split a PSOS such that a deadlock-free execution is possible in the presence of guarded actions semantics, it can also be observed that a PSOS for the overall model may exist. In this case, it seems counterintuitive to implement distributed PSOS, which must be additionally split into multiple transitions in order to not introduce deadlocks into the model. Note that in the general case, the actor FSMs representing these quasi-static schedules may become quite large in practice (cf. Section 5.3.2). The periodic partial order scheduling approach presented in [ZHF+13] solves this problem by providing for a partial ordering of actor firings in contrast to the total ordering imposed by a PSOS. In the proposed dataflow model, this corresponds to a partial ordering of transition sets (tasks) from which an enabled transition is selected at run time (cf. Definition 5.3). Thus, this approach still allows a concurrent execution of actors on different resources, but, in contrast to the distributed PSOS approach, provides for a simple task synchronization mechanism not based on FIFO channels. In principle, this approach therefore also

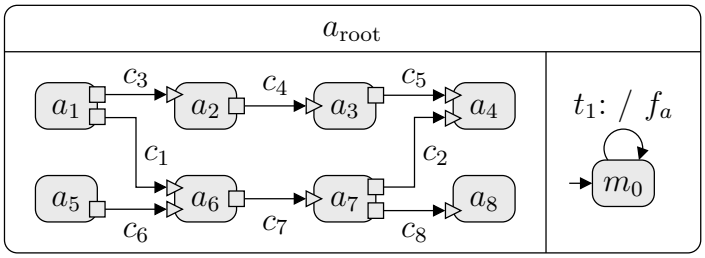


Figure 5.7: The root actor a_{root} and HSDF actors a_1–a_4 are bound to resource r_1, while actors a_5–a_8 are bound to resource r_2. The action function f_a implements a PPOS for actors a_1–a_8.

allows the replacement of inter-resource FIFO channels by simpler communication primitives without a coupled synchronization mechanism, like registers.

Example 5.7. Consider Figure 5.7, which contains the same HSDF leaf actors a_1–a_8 as shown in Figure 5.6. However, the functional composite actors γ_1 and γ_2 are not required in this case, as the root actor a_{root} implements a PPOS for actors a_1–a_8. To this end, a task list $\lambda = (\Gamma, \prec)$ is executed by the action function f_a of the only transition t_1 of a_{root}. The tasks executed by λ correspond to $\Gamma = \{\tau_1 = a_1, \tau_2 = a_2, \ldots, \tau_8 = a_8\}$, and the tasks are ordered as follows: (1) $\tau_1 \prec \tau_2 \prec \tau_3 \prec \tau_4$, (2) $\tau_5 \prec \tau_6 \prec \tau_7 \prec \tau_8$, (3) $\tau_1 \prec \tau_6$, and (4) $\tau_7 \prec \tau_4$. Given the resource binding $\beta(a_{\text{root}}) = \beta(a_1) = \beta(a_2) = \beta(a_3) = \beta(a_4) = r_1$ and $\beta(a_5) = \beta(a_6) = \beta(a_7) = \beta(a_8) = r_2$, this allows, e.g., the concurrent execution of actors a_1 and a_5, a_2 and a_6, etc. Note that all FIFO channels c_1–c_8 could be replaced by registers in this case.

5.3.2 Quasi-Static Scheduling

In the previous section, Example 5.6 illustrated that a given PSOS for a resource might introduce deadlocks into the model, and therefore has to be split into a set of partial PSOS in order to accommodate token feedback loops over multiple actors. The periodic partial order scheduling approach solved this problem differently. However, it is only applicable if an overall PSOS for the actors that are part of a token feedback loop can be determined. In the general case, an application consists of static actors and dynamic actors, and token feedback loops may include dynamic actors. Note that in this context, *dynamic actor* refers to actors that are not CSDF actors (and therefore, neither SDF nor HSDF actors). In this case, an overall PSOS for the relevant actors cannot be determined, but implementing a PSOS only for static subgraphs might be infeasible in the general case as illustrated by Example 5.6.

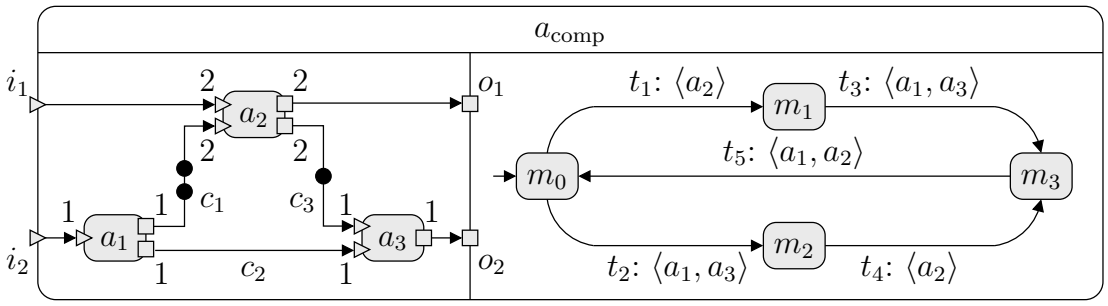


Figure 5.8: QSS for actors a_1, a_2, and a_3 implemented by the functional composite actor a_{comp}. In order to improve readability, token consumption and production rates have been omitted in the actor FSM, and the partial PSOS annotated to each transition is executed by the corresponding action function.

A possible solution could be to resort to the dynamic scheduling of all actors, including the static actors of an application. However, this might introduce unnecessary scheduling overhead for the static actors of the application. Another possibility consists in quasi-statically scheduling the static actors. In this case, the QSS for the static subgraph must be constructed such that the introduction of deadlocks into the model is prevented. Basically, each transition of the actor FSM which implements the resulting QSS then corresponds to a partial PSOS as illustrated by Example 5.6.

The basic idea to derive such a QSS has been presented in [FKH+08; FZK+11] and consists in the observation that the resulting QSS does not introduce deadlocks into the model if each partial PSOS consumes a minimal number of tokens from input ports of the composite actor which implements the QSS, and produces a maximal number of tokens on output ports of the composite actor. In other words, the QSS must not hold back already produced tokens in order to wait for some additional tokens on input ports to produce some more tokens.

Example 5.8. For the functional composite actors depicted in Figure 5.6, the resulting partial PSOS have already been described in Example 5.6: For actor γ_1, PSOS $S_{1,1} = \langle a_1, a_2, a_3 \rangle$ and $S_{1,2} = \langle a_4 \rangle$, while for actor γ_2, the PSOS $S_2 = \langle a_5, a_6, a_7, a_8 \rangle$ does not require splitting.

A more complex example is shown in Figure 5.8. Here, actors a_1, a_2, and a_3 are SDF actors with the given token consumption and production rates. Note that $\mathrm{D}(c_1) = 2$, and $\mathrm{D}(c_3) = 1$, i.e., two initial tokens are placed on channel c_1, while one initial token is placed on channel c_3. A PSOS corresponding to the overall repetition vector $\boldsymbol{\gamma} = (2, 1, 2)$ is, e.g., $S = \langle a_1^2, a_2, a_3^2 \rangle$. However, implementing S by means of a single transition might introduce deadlocks into the model. Instead, we have to compute actor firings such that a maximal

number of tokens is produced on output ports o_1 and o_2 by consuming a minimal number of tokens from input ports i_1 and i_2.

Initially, two possibilities exist: If two tokens are available on input port i_1, actor a_2 can be fired once, thereby producing two tokens on output port o_1. This corresponds to transition t_1 which executes the partial PSOS $S_1 = \langle a_2 \rangle$. On the other hand, one token may be available on input port i_2. In this case, actor a_1 can be fired once, followed by a single firing of actor a_3, which produces one token on output port o_2. This corresponds to transition t_2 which executes the partial PSOS $S_2 = \langle a_1, a_3 \rangle$.

In mode m_1, no tokens are left on channel c_1, while three tokens are available on channel c_3. Thus, actor a_2 cannot be fired even if two tokens are available on input port i_1. Instead, we must wait until one token is available on input port i_2 in order to execute PSOS $S_3 = \langle a_1, a_3 \rangle$ by means of transition t_3.

Analogously, in mode m_2, no tokens are left on channel c_3, while three tokens are available on channel c_1. Thus, actor a_1 cannot be fired even if a token is available on input port i_2. Instead, we must wait until two tokens are available on input port i_1 in order to execute PSOS $S_4 = \langle a_2 \rangle$ by means of transition t_4.

Finally, in mode m_3, one token is available on channel c_1, while two tokens are available on channel c_3. Again, actor a_2 cannot be fired even if two tokens are available on input port i_1. Thus, when a token is available on input port i_2, the PSOS $S_5 = \langle a_1, a_3 \rangle$ is executed by transition t_5. Note that after executing transition t_5, two tokens are available on channel c_1, and one token is available on channel c_2. This corresponds to the initial mode m_0, which is therefore the target mode of transition t_5.

It should be noted that the actor classification algorithm presented in Section 4.3.2 correctly finds a repeatable communication pattern $\mathbf{cp} = (2, 2, 2, 2)$, corresponding to the overall PSOS $S = \langle a_1^2, a_2, a_3^2 \rangle$. However, the subsequent validation of the partitioned communication pattern fails as expected, and the actor is therefore not classified as a CSDF actor.

It can be observed that the resulting actor FSM which implements a QSS derived by the methodology presented in [FKH+08; FZK+11] contains transitions that execute the very same partial PSOS. In order to reduce the size of the actor FSM, [FZHT11; FZHT13] describes how these transitions can be merged in principle. To this end, *rules* are used that encode lower and upper bounds on the number of actor firings for which a certain partial PSOS is enabled, i.e, could be executed if enough tokens and free places are available. Basically, while the original method explicitly encodes the number of tokens on channels of the composite actor by means of modes, the rule-based method encodes this information by counting the number of firings which have already been performed for each child actor. While this requires some additional actor variables and

Rule	Lower bound $\mathbf{l_i}$	Upper bound $\mathbf{u_i}$	Increment $\mathbf{s_i}$	PSOS
r_1	$(0,0,0)$	$(0,\infty,0)$	$(1,0,1)$	$S_1 = \langle a_1, a_3 \rangle$
r_2	$(0,0,0)$	$(\infty,0,\infty)$	$(0,1,0)$	$S_2 = \langle a_2 \rangle$
r_3	$(1,1,1)$	$(1,\infty,1)$	$(1,0,1)$	$S_3 = \langle a_1, a_3 \rangle$

Table 5.2: Rules generated for quasi-statically scheduling the actors a_1, a_2, and a_3 from Figure 5.8. Note that the repetition vector $\boldsymbol{\gamma} = (2,1,2)$.

induces some additional run-time overhead, the overall size of the generated actor FSMs is reduced.

Example 5.9. Consider again Figure 5.8. Applying the rule-based quasi-static scheduling approach results in three rules r_1, r_2, and r_3 summarized by Table 5.2. As there are three SDF actors to be scheduled, the lower and upper bounds on the number of actor firings are encoded by vectors $\mathbf{l_i} \in \mathbb{N}_0^3$ and $\mathbf{u_i} \in (\mathbb{N}_0 \cup \{\infty\})^3$. Let $\mathbf{q} \in \mathbb{N}_0^3$ denote the number of actor firings which have already been performed. Initially, $\mathbf{q} = \mathbf{0}$. Then, a rule r_i is enabled if $\mathbf{l_i} \leq \mathbf{q} \leq \mathbf{u_i}$. For $\mathbf{q} = \mathbf{0}$, both rules r_1 and r_2 are enabled, corresponding to transitions t_1 and t_2 in Figure 5.8.

When a rule r_i is executed, the corresponding PSOS is executed, and $\mathbf{q}$ is incremented by the value of $\mathbf{s_i}$., i.e., $\mathbf{q} \leftarrow \mathbf{q} + \mathbf{s_i}$. For example, if r_1 is executed, PSOS $S_1 = \langle a_1, a_3 \rangle$ is executed, and $\mathbf{q} \leftarrow (0,0,0) + (1,0,1) = (1,0,1)$. Note that now, only rule r_2 is enabled, corresponding to transition t_4 in Figure 5.8. If r_2 is executed instead of r_1, $\mathbf{q} \leftarrow (0,1,0)$, and only rule r_1 remains enabled, corresponding to transition t_3 in Figure 5.8.

When both rules r_1 and r_2 have been executed, it follows that $\mathbf{q} = (1,1,1)$, and rule r_3 becomes enabled, corresponding to transition t_5 in Figure 5.8. Finally, when r_3 has been executed, it follows that $\mathbf{q} = (2,1,2)$. In this case, $\mathbf{q} \geq \boldsymbol{\gamma} = (2,1,2)$, and we set $\mathbf{q} \leftarrow \mathbf{q} - \boldsymbol{\gamma} = (0,0,0)$.

In the proposed dataflow model, rules are implemented by self-loop transitions of the initial mode. Note that only two transitions are required in this case, as r_1 and r_3 execute the same PSOS, and also have the same token consumption and production rates. While the lower and upper bounds $\mathbf{l_i}$ and $\mathbf{u_i}$ are evaluated by the guard function of a transition, the action function executes the corresponding PSOS by means of a task list, increments $\mathbf{q}$ by $\mathbf{s_i}$, and finally subtracts $\boldsymbol{\gamma}$ from $\mathbf{q}$ if $\mathbf{q} \geq \boldsymbol{\gamma}$.

Finally, it should be noted that an FSM which implements a QSS derived by the approaches described in this section can be easily distributed across multiple resources by means of the periodic partial order scheduling approach described in Section 5.3.1.

5.3.3 Static-Assignment Scheduling

Static-assignment scheduling (or dynamic scheduling) basically encompasses any scheduling strategy that can be implemented by the proposed approach. Due to the black box approach for guard functions and action functions, a wide range of dynamic scheduling strategies can be implemented. In particular, a task τ may contain only a single transition t, in which case the evaluation and execution of τ corresponds to the evaluation and execution of the guard function $t.f_g$ and the action function $t.f_a$, providing for a fine-grained selection of transitions of child actors.

Table 5.3 summarizes the scheduling strategies which are currently supported by the proposed dataflow model. Compared to Table 5.1, it can be observed that a dynamic binding of actors to resources is currently not supported. Moreover, non-functional scheduling schemes like time-driven scheduling or preemptive scheduling are not considered in this thesis. Typically, such scheduling schemes are implemented by the RTEs of resources (cf. Section 5.2.2). In this context, the integration of non-functional scheduling schemes could be studied by future work.

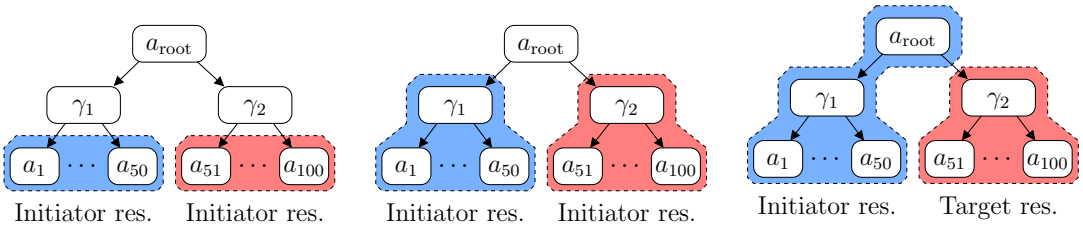
5.4 Results

In this section, we quantitatively compare the scheduling strategies discussed in the previous section (cf. Table 5.3). To this end, we generated synthetic SDF graphs and mapped them to different multicore processors. The structure of the generated SDF graphs is comparable to the DFG shown in Figure 5.6. Inspired by real-world multimedia applications, the actors bound to each resource could represent, e.g., the per-component encoding/decoding functionality of audio/video streams (cf. Figure 5.2) on a fine-grained (e.g., macroblock) level. Note that synthetic SDF graphs have been chosen in order to systematically explore the influence of various graph properties like token consumption and production rates on the evaluated scheduling strategies.

As the focus of this chapter is not how to find (Pareto-)optimal binding and scheduling solutions of arbitrary SDF graphs, but rather to show how such binding and scheduling decisions can be represented in a model-based manner,

Scheduling class	Scheduling task		
	Assignment	Ordering	Timing
Static-assignment	Compile	Run	Run
Quasi-static	Compile	Run/Compile	Run
Self-timed	Compile	Compile	Run

Table 5.3: Scheduling strategies supported by the proposed dataflow model.



a) Testcase *DS* b) Testcases *PSOS* and *QSS* c) Testcase *PPOS*

Figure 5.9: Actor hierarchy and resource types as induced by the binding of actors to resources (indicated by the colored areas).

we generated acyclic single-rate SDF graphs (cf. Section 4.2) in the following way: The application is represented by the root actor a_{root}, which instantiates a composite actor γ_i for each resource r_i. In turn, each composite actor γ_i instantiates the same number of SDF actors, resulting in approximately the same latency w.r.t. each core, thereby minimizing the overall latency of the DFG, which could be an optimization objective of design space exploration. Note that we chose 50 actors per composite actor, representing a fine-grained DFG. However, given the nature of the generated graphs, the latency scales linearly with the number of actors. The resulting actor hierarchy $G_A = (V_A, E_A)$ is shown in Figure 5.9. After instantiating the actors, channels are added between the SDF actors such that the resulting graph is acyclic, thus enabling a deadlock-free execution of the graph without having to add initial tokens.

A prototype of the modeling framework has been implemented in C#, which offers extensive support for reflection, which is important for synthesis tasks. The framework allows DFGs to be created and transformed according to the proposed dataflow model. However, there is no simulation built into the model. Instead, model transformations as well as simulation and synthesis tools can be implemented by means of *plug-ins* which process a given DFG. To this end, we implemented a functional simulation plug-in (also in C#), which performs the binding-aware evaluation and execution phases as described in Section 5.2. In particular, for each resource, a single thread is created. Note that the functional simulation of the model mostly corresponds to the software synthesis of the model, which is discussed in more detail in Chapter 6. The following testcases have been evaluated:

- Testcase *DS* dynamically schedules the SDF actors. In this case, each resource r_i is an initiator resource which schedules its bound SDF actors according to Algorithm 5.7 (cf. Figure 5.9a). This testcase is evaluated for the single-core and multi-core bindings.

- Testcase *PSOS* statically schedules the SDF actors as described in Section 5.3.1. To this end, each composite actor γ_i implements the corre-

sponding PSOS. Each resource r_i is also an initiator resource, but now only dynamically schedules its bound functional composite actor γ_i (cf. Figure 5.9b). This testcase is evaluated only for the single-core bindings.

- Testcase *QSS* quasi-statically schedules the SDF actors as described in Section 5.3.2. To this end, each composite actor γ_i implements the corresponding QSS. Each resource r_i is also an initiator resource, but now only dynamically schedules its bound functional composite actor γ_i (cf. Figure 5.9b). This testcase is evaluated only for the multi-core bindings.

- Testcase *PPOS* implements a periodic partial order schedule as described in Section 5.3.1. To this end, the transitions of each composite actor γ_i first encapsulate the firings of the SDF actors. Then, the root actor a_{root} implements a PPOS for the transitions of the composite actors γ_i. Now, the only initiator resource corresponds to the resource where the root actor a_{root} is bound to (we used r_1 for this task), and the remaining resources are target resources (cf. Figure 5.9c). Note that in this case, only the single transition of a_{root} implementing the PPOS has to be scheduled dynamically. This testcase is evaluated only for the multi-core bindings. Note that a PPOS where all actors are bound to the same resource basically corresponds to a PSOS, which is handled by a separate testcase.

The testcases have been evaluated for two target platforms, namely a) the *Xilinx Zynq-7000 All Programmable SoC* with a dual-core ARM Cortex-A9 processor operating at 667 MHz, and b) an Intel Xeon E7-8837 processor with 8 cores operating at 2.67 GHz. Both platforms are running Linux and the Mono framework [Mon14], which is an open source implementation of the ECMA standard for C# [Ecm06]. In case of a multi-core binding, thread-safe FIFO channels are used if necessary, and each thread is pinned to the corresponding processor core in order to minimize the influence of the operating system.

The simulation results are shown in Figure 5.10, 5.11, and 5.12. For each testcase, several graph parameters have been explored as explained in the following:

- *Avg. out degree* refers to the average number of output ports per actor. This parameter also corresponds to the average number of input ports per actor.

- *Avg. computation* refers to the average number of operations (additions) performed by each actor. The single-core testcases have been evaluated for computation values of 100 and 1000. For the multicore testcases, the computation value is fixed at 1000.

- *Avg. communication* refers to the average token consumption and production rates of actors w.r.t. the input and output ports. All testcases have been evaluated for rates of 1, 5, 10, and 20 tokens.

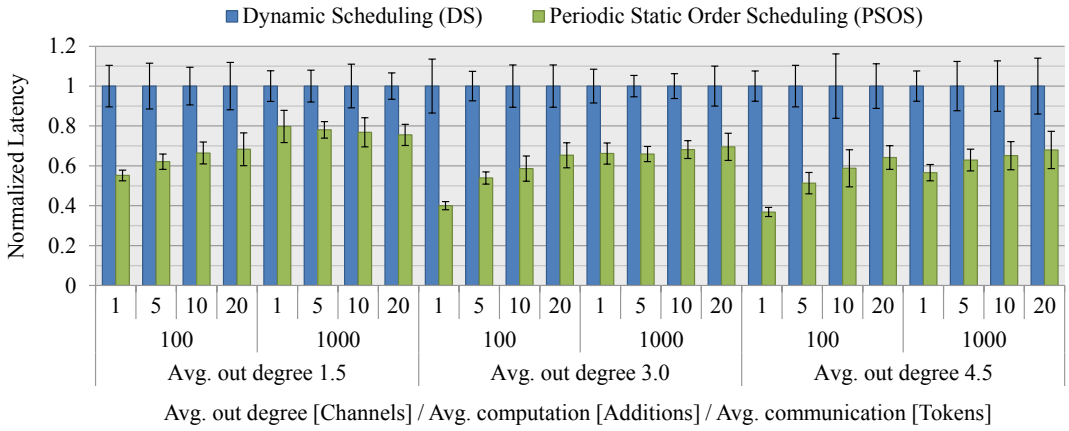
- *Inter-resource channels* refers to the fraction of channels which connect actors that are bound to different resources w.r.t. the total number of channels. The multi-core testcases have been evaluated for 20% and 50% of all channels being inter-resource channels. As intra-resource communication is usually preferred to inter-resource communication, larger values have not been considered.

In order to show the impact of the scheduling strategies on the scheduling overhead, the latency values of the evaluated testcases are normalized against the *DS* testcase. For each set of parameter values, 10 random DFGs have been generated, and each DFG has been evaluated 5 times. In order to obtain meaningful results, the number of graph iterations has been chosen such that execution times of several seconds per graph evaluation have been obtained. For each set of parameter values, the given (normalized) latency is therefore the average latency of 50 graph evaluations. Note that besides the average latency, the standard deviation is shown.

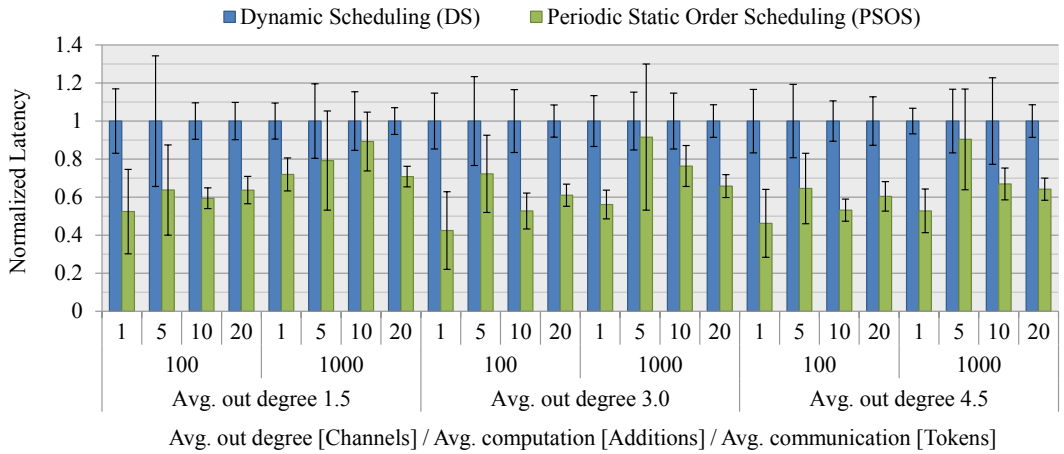
Concerning the resulting latencies of the single-core bindings (cf. Figure 5.10), it can be observed that *PSOS* always performs better than *DS*, in particular for smaller values of the computation and communication parameters. This stems from the fact that if the scheduling overhead remains constant, its influence on the overall latency is diminished when action functions take longer to complete (which is the case for larger values of the computation and communication parameters): Assume that L_{DS} and L_{PSOS} denote the measured latencies for a given set of parameter values, and that $L_{\mathrm{DS}} < L_{\mathrm{PSOS}}$. The normalized *PSOS* latency is therefore $L = \frac{L_{\mathrm{PSOS}}}{L_{\mathrm{DS}}}$. Increasing only the values of the computation and communication parameters, the same latency increase L_Δ should be observed for both testcases in principle, corresponding to additional arithmetic operations and memory accesses performed by action functions. Then, it follows that the resulting normalized *PSOS* latency $L' = \frac{L_{\mathrm{PSOS}}+L_\Delta}{L_{\mathrm{DS}}+L_\Delta} > L$ (without proof).

If the graph structure is modified by increasing the average number of input/output ports per actor, the scheduling overhead imposed by *DS* also increases, as increasing the number of channels means that more token and space availability checks must be performed. Note that the scheduling overhead imposed by *PSOS* can be considered constant and is very small, as each composite actor γ_i only consists of a single transition which is dynamically scheduled by the corresponding RTE. In particular, token and space availability checks can be eliminated in this case.

In case of the ARM platform, these effects are well observable. In case of the Xeon platform, the latency difference between *DS* and *PSOS* for graphs with average token consumption and production rates of 5 and 10 is less than expected. This may be caused by the more complex cache hierarchy of the Xeon



a) ARM Cortex-A9 (1 core)

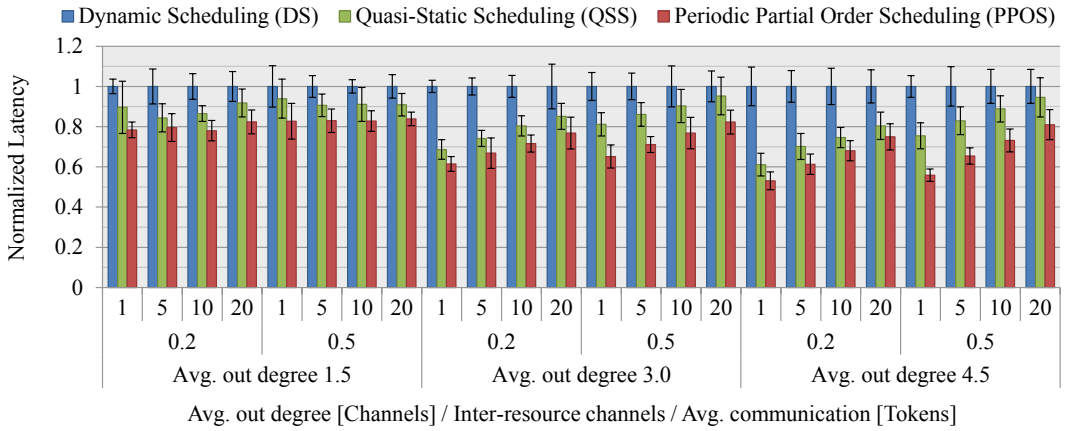


b) Intel Xeon E7-8837 (1 core)

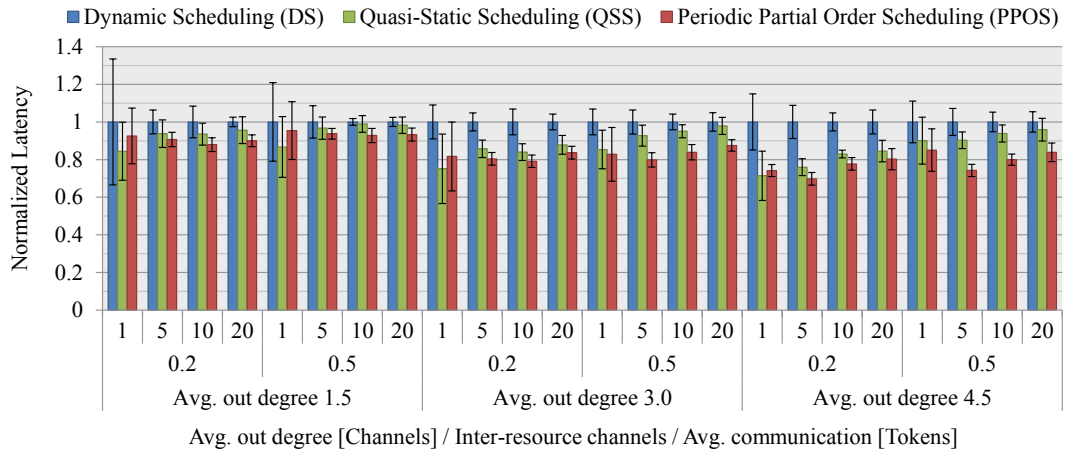
Figure 5.10: Single-core simulation results comparing the scheduling strategies in terms of latency. The results are normalized against the *DS* testcase.

processor, which typically causes a highly dynamic behavior, and makes latency predictions difficult in the general case [WM05].

Concerning the resulting latencies of the multi-core bindings (cf. Figure 5.11), the same observations can be made for the *QSS/PPOS* testcases versus the *DS* testcase in principle. However, it can be observed that the latency difference between *QSS/PPOS* and *DS* is smaller compared to the single-core testcases. In this case, the dynamic scheduling performed by the RTE of each resource seems to be advantageous compared to the static actor order imposed by the *QSS/PPOS* testcases, which possibly prevents enabled actors from being executed earlier on an idle resource.



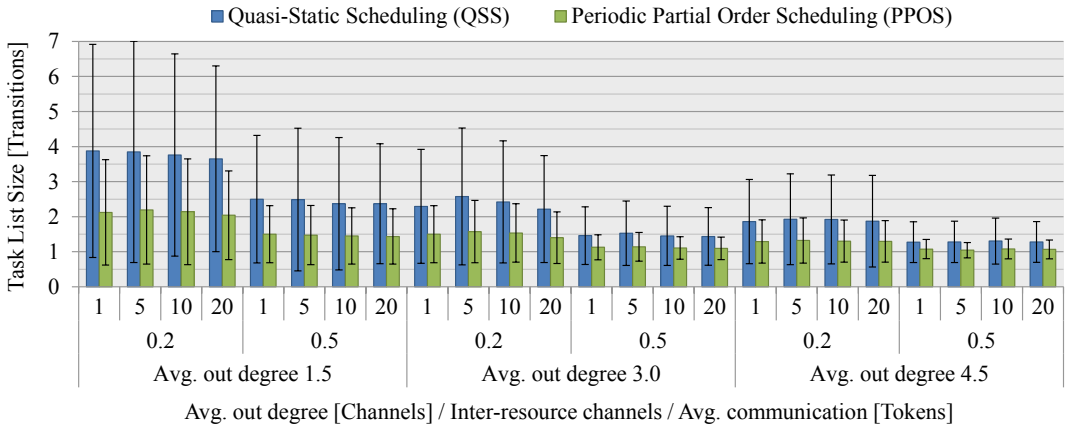
a) ARM Cortex-A9 (2 cores)



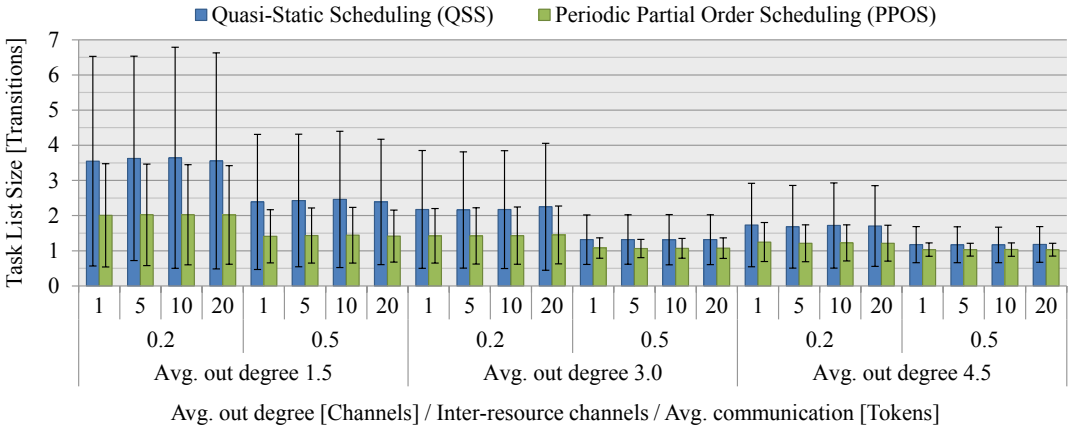
b) Intel Xeon E7-8837 (8 cores)

Figure 5.11: Multi-core simulation results comparing the scheduling strategies in terms of latency. The results are normalized against the *DS* testcase. The computation value is fixed at 1000 for all testcases.

Furthermore, it can be observed that *PPOS* outperforms *QSS* in almost all testcases. Remember that periodic partial order scheduling provides for a more efficient synchronization mechanism compared to quasi-static scheduling. In particular, the implementation uses a single bit to signal the completion of a task τ to the successor tasks of τ which are executed on different resources. Assuming that the selected transition $t \in \tau$ produces tokens on multiple inter-resource channels, evaluating this single bit may subsume multiple token availability checks which would have to be performed by the successor tasks of τ in the *QSS* scenario. Analogously, multiple space availability checks may be subsumed by evaluating this single bit. Thus, the larger the number of ports per actor bound



a) ARM Cortex-A9 (2 cores)



b) Intel Xeon E7-8837 (8 cores)

Figure 5.12: Multi-core simulation results comparing the scheduling strategies in terms of the resulting task list sizes of the composite actors γ_i. The computation value is fixed at 1000 for all testcases.

to inter-resource channels becomes, the larger the latency difference between *QSS* and *PPOS* becomes.

However, it should be noted that even in the *DS* and *QSS* scenarios, token and space availability checks can be performed very efficiently once the read and write pointers of the FIFO channel in question are cached. Moreover, the cache coherence protocol which is used by the ARM and Xeon processors ensures that the cached copies of the read and write pointers of the FIFO channels are efficiently updated when these pointers are modified by the consumption and production of tokens. Thus, for the considered multi-core processors, the latency difference between *QSS* and *PPOS* can be expected to be relatively small.

In Figure 5.12, the average size of the task lists executed by the transitions of the composite actors γ_i is shown. Note that only results for the multi-core bindings are given, as in case of the single-core bindings, the task list executed by the single transition of γ_1 corresponds to the overall PSOS, and thus obviously contains all 50 SDF child actors of γ_1. It can be observed that the larger the number of ports per actor bound to inter-resource channels becomes, the smaller the average size of the task lists becomes. For the *QSS* testcase, this follows directly from the splitting of the overall PSOS into partial PSOS in order to prevent the introduction of deadlocks into the model (cf. Example 5.6). Concerning the *PPOS* testcase, a similar scheme has been used which results in slightly shorter partial PSOS in order to minimize the impact on the resulting latencies. Note that in this case, however, the composite actors γ_i could have been eliminated instead (cf. Example 5.7), as the overall PPOS is implemented by the root actor a_{root}.

For both testcases (i.e., *QSS* and *PPOS*), the size of the task lists has little impact on the overall performance: Concerning the *QSS* testcase, the resulting FSMs basically show CSDF behavior, i.e., the partial PSOS are executed in a cyclic fashion (cf. Figure 4.5 on page 57). Thus, the scheduling overhead in order to select the next transition to be executed is negligible in this case (as each mode only has a single outgoing transition). Concerning the *PPOS* testcase, the scheduling overhead is minimized anyway, as the overall PPOS is implemented by the root actor a_{root}. Remember that in this case, only the single transition of a_{root} implementing the PPOS has to be scheduled dynamically.

It should be noted that the results show mostly the predicted behavior. However, it was not our goal to determine which scheduling strategy is the best for a given DFG, but to show that the proposed hierarchical modeling approach is able to effectively and efficiently represent different scheduling strategies in a model-based way.

5.5 Related Work and Limitations

The refinement of models in order to incorporate binding and scheduling decisions has been studied before. The approaches can roughly be divided into three categories, namely *analysis-oriented* approaches, *implementation-oriented* approaches, and *simulation-oriented* approaches. Analysis-oriented approaches try to retain the analyzability of the original dataflow model. To this end, binding and scheduling decisions are typically incorporated into the underlying dataflow model solely by modeling primitives provided by the dataflow model of computation (MoC) (like SDF or CSDF) corresponding to the underlying dataflow model. In contrast to analysis-oriented refinement approaches, implementation-oriented approaches trade off analyzability against the possibility to derive efficient implementations from the refined model. Note that the proposed refinement approach

based on hierarchical actors belongs to this category. Finally, simulation-oriented approaches typically possess an inherent notion of time, and are therefore used as virtual platforms for software development, and for performance analysis of hardware architectures.

Analysis-oriented approaches try to retain the analyzability of the original dataflow model. The decision state modeling (DSM) technique presented in [DSB+12] back-annotates a PSOS for an SDF (sub)graph by adding channels to the original SDF graph such that i) subsequent firings of the same actor are serialized, (ii) subsequent iterations of the PSOS are serialized, and (iii) the actor ordering imposed by the PSOS is enforced. While (i) is realized by adding a self-loop channel to each actor appearing in the PSOS, (ii) is realized by adding an actor and two channels to create a dependency between the last actor and the first actor appearing in the PSOS. Finally, (iii) is realized by forcing the correct actor firing w.r.t. the PSOS in the *decision states*. In a decision state, more than one actor appearing in the PSOS is enabled in the underlying DFG. Therefore, in order to select the only permissible actor according to the PSOS, additional channels are added. The resulting SDF graph can then be used to apply well-known SDF analysis techniques, like latency and throughput calculation, or buffer sizing. This approach can only be used for DFGs for which a PSOS can be determined, i.e., static graphs like SDF and CSDF graphs. In contrast, the proposed implementation-oriented approach aims at dynamic dataflow (DDF) graphs and thus also supports, for example, quasi-static scheduling (which requires run-time decisions) and the switching of scheduling modes as known from scenario-aware dataflow (SADF). The DSM technique has been extended in [LF13] in order to incorporate binding and pipeline stage assignment decisions.

In [BKKB02], PSOS are back-annotated by first translating the underlying SDF graph into a functionally equivalent HSDF graph, and then adding channels to the resulting HSDF graph such that the actor ordering imposed by the PSOS is enforced. As outlined in Section 4.2, the number of actors of the HSDF graph grows exponentially with the number of actors of the SDF graph in the worst case. Thus, this approach is typically only feasible for small SDF graphs. Note that this problem is avoided by the DSM technique presented in [DSB+12]. Besides the back-annotation of PSOS, [BKKB02] also introduces the *ordered transactions* scheduling strategy. The ordered transaction scheduling strategy requires that in addition to the PSOS for each resource, the actor firings which perform inter-resource communication are also totally ordered. The resulting ordering of actor firings is then enforced by special hardware. As a result, the ordered transaction scheduling strategy eliminates the need for run-time synchronization and bus arbitration. Note that the periodic partial order scheduling approach presented in Section 5.3.1 is not concerned about the total ordering of transactions on a bus, but provides for the partial ordering of transitions of actors bound to different

resources. While an ordered transaction schedule could be represented by a PPOS by totally ordering the transitions of actors which perform inter-resource communication by means of the same communication resource, the synthesis of special hardware for the communication resource is not addressed in this thesis, but could be part of future work.

Implementation-oriented approaches trade off analyzability against the possibility to derive efficient implementations from the refined model. Here, one possibility consists in incorporating the binding and scheduling decisions by means of an unspecified internal representation, and to output only the resulting model in a high-level language like C/C++. While this approach has been traditionally used to synthesize DFGs [BL93; Buc94; BBHL95; BLM96; BML97; SLWS99; BLM00; HKB05; KB06], the analysis of the refined model is difficult. In order to retain some analyzability, more recent approaches incorporate binding and scheduling into the model by means of hierarchical DFGs, comparable to the proposed dataflow model. To this end, the firing of an actor is usually refined by firings of other actors.

The generalized schedule tree (GST) approach [KZP+07] differentiates between leaf nodes and internal nodes: While leaf nodes represent actors from the underlying DFG, internal nodes implement looped schedules, i.e., they execute its child nodes a given number of times. The underlying DFG is based on enable-invoke dataflow (EIDF), which has been reviewed in Section 3.4. In this context, EIDF actors cannot specify schedules for child actors. Instead, a secondary scheduling mechanism (like GSTs) is required. In contrast, actors in our model are expressive enough to specify schedules for child actors, thereby eliminating the need for secondary scheduling mechanisms. In [PSB09b], it is described how a GST can be decomposed into a set of static interacting graphs in order to improve the simulation time compared to a dynamic execution of the GST.

In [WSS+11], dataflow schedule graphs (DSGs) are used to implement schedules for actors of an underlying DFG. A DSG consists of *reference actors* and *schedule control actors*. While the former are used to represent firings of actors of the underlying DFG, the latter are used to model control-flow as known from sequential programming languages (e.g., loops and branches). In order to guarantee a sequential execution of actors bound to the same resource, a single control-flow token is passed along reference and schedule control actors. Additionally, control flow tokens may be duplicated and merged to model synchronization between concurrent resources. As dataflow actors and reference actors are only loosely coupled, the authors claim that the approach works for any underlying dataflow representation based on guarded actions, e.g., EIDF. Again, the need for a secondary scheduling mechanism like DSGs is eliminated by the proposed refinement approach, as composite actors are expressive enough to specify schedules for child actors. Note that control flow in the proposed dataflow model is naturally represented by FSMs.

FunState [TSZ+99; STG+01] uses nested components which are controlled by FSMs (cf. Section 3.4). As has been pointed out, action functions use *events* to activate nested components. This may result in a non-sequential behavior of components, where a transition may be started before the previous one is finished. In turn, actor variables must be stored in a self-loop channel, and tokens must be consumed and produced atomically at the beginning and end of an action, respectively. In contrast, transitions of an actor have sequential semantics in the proposed dataflow model, and thus, stateful actors are permitted, and tokens can be consumed and produced at any time during the execution of a transition. Moreover, transitions of the proposed dataflow model have a static communication behavior, which may not be the case in FunState.

The structural composite actors available in SysteMoC (cf. Section 3.4) have been extended in order to support the rule-based quasi-static scheduling approach described in Section 5.3.2 [FZHT11; FZHT13]. Basically, a counter is associated with each actor which is incremented when the actor is fired. In turn, a firing rule encodes lower and upper bounds on these counters to determine when it is enabled. Thus, these firing rules have limited expressiveness compared to guard functions and action functions of the proposed dataflow model.

Finally, formalized simulation-oriented approaches like [KR11] possess an inherent notion of time, and are therefore used as virtual platforms for software development, and for performance analysis of hardware architectures. As a purely functional model, our proposed approach is architecture-independent, but in principle can be annotated with timing information in order to obtain a transaction-level performance model (cf. [SFH+06; SGHT09]).

The ForSyDe methodology presented in [SJ04] is based on a synchronous MoC. The synchronous computational model is implemented by means of *process constructors*. A process constructor is a higher-order function that takes combinational functions and values as input, and produces a process as output. Each process constructor explicitly specifies hardware and software semantics which provides for the synthesis of processes to lower levels of abstraction. In contrast to the proposed dataflow model, the designer is restricted to a set of pre-defined process constructors in order to specify a system. Model refinement is performed by *transformation rules* which can be classified into *semantic preserving transformations* and *design decisions*. While the former are mainly used to optimize the model, the latter change the meaning of the model. Transformation rules are also pre-defined and are chosen from a *transformation library*. In [AS11], it is described how processes can be refined by components that are modeled and executed in different tools and languages. This approach is employed in [BAS12] in order to refine processes by SystemC-transaction level modeling (TLM) components. In principle, this refinement approach provides for the heterogeneous co-simulation of a partially refined system. For the proposed dataflow model,

a similar co-simulation approach could be realized by means of the synthesis framework presented in the next chapter.

6 Hardware/Software Synthesis

After a fully bound and scheduled model has been obtained, the next step in the design flow could consist of synthesizing the model to the given target architecture. For actors bound to software, this requires the generation of code which is executed on the target processors, while for actors bound to hardware, custom intellectual property (IP) cores must be generated (if not available off-the-shelf).

Figure 6.1 summarizes the proposed design flow from the algorithmic/task level to the logic/instruction level. Based on the input model, namely a scheduled dataflow graph (DFG) which has been obtained as described in Chapter 5, Section 6.1 describes a framework which provides for the synthesis of the input model such that arbitrary bindings of actors to hardware and software are supported. Section 6.2 describes the task of communication synthesis at the algorithmic/task level in the context of the proposed synthesis framework. Analogously, Section 6.3 discusses the task of computation synthesis, and describes

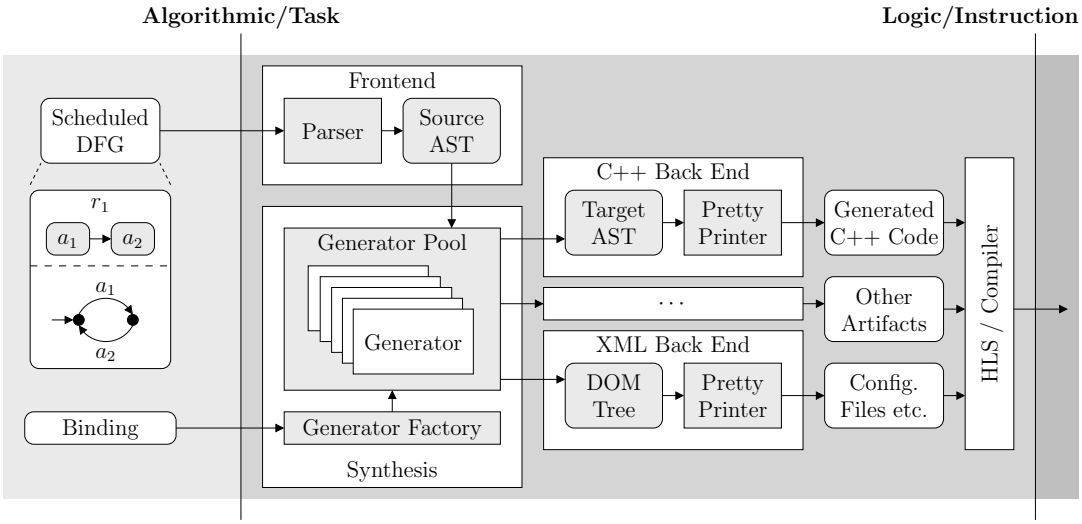


Figure 6.1: Design flow supported by the proposed model. Only the steps pertaining to behavioral synthesis are shown (cf. Figure 2.2 on page 9).

optimizations and an inter-process resource sharing approach in the context of hardware synthesis. Finally, Section 6.4 discusses related work and the limitations of the hardware/software synthesis approach.

6.1 Synthesis Framework

Based on the dataflow model described in Sections 3.1 and 5.1, the proposed synthesis framework has been introduced in [ZHFT12b]. Please note that the proposed framework does not make any assumptions about the binding of actors to resources (hardware/software, or mixed).

At a glance, the synthesis framework processes an input dataflow model in three phases as shown in Figure 6.1: i) The front end parses the source code of the input model, and provides the resulting abstract syntax tree (AST) to the subsequent synthesis phase. ii) During the synthesis phase, the different components of the scheduled DFG are processed by *generators* in order to generate all necessary data for the next phase. Generators are instantiated by a *generator factory* based on the binding of actors to resources. iii) The various back ends collect the data generated during the previous phase, and finally assemble all files which make up the synthesized application. In the following, the three phases are discussed in more detail.

6.1.1 Front End

The task of the front end is to extract static information from the scheduled DFG. This static information corresponds to information which can be determined at compile time, like guard functions and action functions of actors, token types, etc. In the general case, this information is represented by an AST. For the C# implementation of the proposed dataflow model, this information can be conveniently extracted from the compiled application by means of the reflection system built into C#. Note that in this case, guard functions and action functions have been compiled into the Common Intermediate Language (CIL). For a C++ implementation of the proposed dataflow model, this information can be extracted, e.g., by the LLVM/Clang modular compiler framework [LA04], which provides access to the C++ AST in an object-oriented way. The static part of the application can be used by generators, for example, to extract token sizes, or to perform source code transformations of guard functions or action functions.

The various components of the DFG (like actors, ports, channels, etc.) cannot be statically extracted by a parser, as we allow these components to by dynamically constructed during an *elaboration phase*. This allows the user, for example, to procedurally generate the transitions during elaboration, or to instantiate actors based on some configuration parameters. Note that the DFG is assumed to be

static after this elaboration phase. In particular, task lists (cf. Section 5.1.1) are currently also supposed to be constructed during the elaboration phase. However, different task lists can be evaluated and executed at run time depending on the values of actor variables or tokens. The C# implementation of the proposed dataflow model permits to query the components of the generated DFG after this elaboration phase. Future work may research the possibility of dynamically instantiating child actors in order to provide for an improved utilization of resources available in the platform [THH+11; HHB+12].

6.1.2 Synthesis

For each component of the DFG, exactly one *generator* exists which synthesizes this particular element. Basically, a generator is associated with a specific component of the DFG (an actor, a port, a channel, etc.) and transforms this element into the target representation. For example, the actor's target representation could be a C++ class, in which case the generator would create a corresponding C++ AST snippet.

Generators may require information from other generators. In order to be able to mix generators for different synthesis targets, generators implement a certain interface corresponding to their associated dataflow component. For example, the generator associated with a guard function or action function can be queried to return the name of the synthesized function. This name can then be used, e.g., by the generator responsible for synthesizing a transition t to generate the function calls to $t.f_g$ and $t.f_a$. As can be seen from this example, generators usually have state. Due to this reason, there exists only one generator for a specific element during the whole synthesis process. This is enforced by the *generator pool* which acts as a generator cache: The pool can be queried for a generator for a given component of the DFG. If the generator already exists, it is returned. If it is not contained in the generator cache, it is created by the *generator factory* (and subsequently inserted into the generator cache).

The creation of generators is the task of the generator factory: Based on the binding of actors to resources as specified by the user (possibly during design space exploration), the factory instantiates the corresponding generator. For example, if a certain actor is bound to a processor, the factory instantiates a generator which performs software synthesis for this actor. However, if the actor should be implemented, e.g., as a transaction level modeling (TLM) module for a virtual prototype, the factory instantiates a different generator. As the exact type of generator is not needed by other generators (the interface methods should be used), this approach provides for the mixing of generators for different synthesis targets in principle. For example, a functional composite actor may be bound to software, while its child actors are bound to hardware.

Given the hierarchical definition of actors (cf. Definition 5.1), components may contain other components (e.g., an actor contains input and output ports). Without going into details, this induces a similar concept for generators, i.e., a generator may be *instantiated* by another generator, possibly modifying the instantiating generator in the process. For example, an actor generator instantiates a port generator for each of its ports. In turn, a port generator may add a variable to the code snippet generated by the actor generator which provides the channel access methods synthesized by the port generator (cf. Section 6.2).

Finally, the *platform generator* is the top-level generator in the hierarchy representing the overall synthesized application. It basically instantiates the generator for the root actor a_{root}, which in turn instantiates the generators for its child actors, and so on. In this way, the DFG is traversed, and the application is synthesized in the process.

6.1.3 Back Ends

The data produced by generators in the previous step is collected by various back ends in order to assemble the files which make up the synthesized application. For generators which produce C++ AST snippets, these are combined into a C++ AST and pretty-printed into one or more source files to be processed by subsequent synthesis steps. For this task, the LLVM/Clang compiler framework can be used. Other back ends are available or can be implemented within the framework in order to produce, e.g., XML configuration files for virtual prototypes or VHDL/Verilog files for the subsequent logic synthesis. Afterwards, the generated artifacts are further processed by a commercial high-level synthesis (HLS) tool or a compiler in order to obtain an implementation at the logic/instruction level.

6.2 Communication Synthesis

The task of computation synthesis at system level is to bind actors to computing resources like processors or custom IP cores, and to schedule actors bound to the same computing resource (cf. Chapter 5). Analogously, communication synthesis at system level comprises the following two tasks: First, the abstract first in, first out (FIFO) channels must be bound to *storage resources* like memories or custom IP cores. In the general case, this induces a scheduling problem if multiple FIFO channels are bound to the same storage resource [MM08]. Second, token sequences must be transported from the computing resources where the actors are bound to the storage resources where the FIFO channels are bound (and vice versa). These *transactions* must be bound to *communication resources* like buses, networks on chip (NoCs), etc. Again, this induces a scheduling problem if multiple transactions are bound to the same communication resource.

Note that the decision-making process itself as part of design space exploration is not discussed in the context of this thesis. The refinement step at system level, i.e., the incorporation of binding and scheduling decisions into the dataflow model has been addressed in Chapter 5 for computing resources to which the actors are bound to. For storage resources and communication resources, similar scheduling strategies can be identified in principle. To some extend, the periodic partial order scheduling approach described in Section 5.3.1 permits to schedule transactions bound to the same communication resource in a model-based way. However, more comprehensive model-based representations of scheduling strategies for storage resources and communication resources may be researched in future work.

In the following, the communication synthesis at the algorithmic/task level is discussed. At this level of abstraction, the two tasks comprising the communication synthesis can be stated as follows: First, depending on the binding of FIFO channels to storage resources, custom IP cores must be generated (if not available off-the-shelf). Second, depending on the binding of transactions to communication resources, appropriate token transport mechanisms must be generated. In the context of the synthesis framework outlined in Section 6.1, *channel generators* perform the first task, while *port generators* perform the second task. Both generator types are described in more detail in the following.

As outlined in Section 3.1, actors communicate with each other by means of token sequences transmitted over FIFO channels. While guard functions evaluate sequences of input tokens (cf. Definition 3.3), action functions transform sequences of input tokens into sequences of output tokens (cf. Definition 3.4). One possibility to realize this concept is to allow guard functions and action functions to randomly access the tokens on FIFO channels within the range specified by the values of peek, cons, and prod, respectively. The resulting abstract channel interface methods are shown in Figure 6.2. The semantics of the write interface provided by output ports can be summarized as follows:

- The `write` method allows an action function $f_a \in F_a$ to write a token on the FIFO channel bound to an output port $p \in O$ within the range specified by the value of $\text{prod}(p, f_a)$. While the token corresponds to the `data` argument, the `offset` argument identifies the token address relative to the current value of the write pointer `wr` associated with the FIFO channel bound to p. Note that the designer must ensure that $0 \leq \texttt{offset} < \text{prod}(p, f_a)$. Otherwise, the model behavior is undefined. The absolute token address is defined as $\texttt{address} = (\texttt{wr} + \texttt{offset}) \bmod \text{K}(p)$.

- The `space` method returns the number of free places on the FIFO channel bound to an output port $p \in O$.

- The `commit` method increments the write pointer `wr` by the value of the `tokens` argument. In effect, this makes some more tokens visible

Figure 6.2: Abstract FIFO channel interfaces provided by input/output ports. Concrete implementations of the interface methods are created by port generators, while FIFO channels are synthesized by channel generators.

at the read interface of the FIFO channel. Note that this operation is performed by the runtime environment (RTE) after an action function has been executed (cf. Algorithm 3.2).

Analogously, the semantics of the read interface provided by input ports can be summarized as follows:

- The `read` method allows a guard function $f_g \in F_g$ or an action function $f_a \in F_a$ to read a token on the FIFO channel bound to an input port $p \in I$ within the range specified by the value of $\text{peek}(p, f_g)$ or $\text{cons}(p, f_a)$, respectively. The `offset` argument identifies the token address relative to the current value of the read pointer `rd` associated with the FIFO channel bound to p. Note that in case of a guard function $f_g \in F_g$, the designer must ensure that $0 \leq \texttt{offset} < \text{peek}(p, f_g)$. Analogously, for an action function $f_a \in F_a$, the designer must ensure that $0 \leq \texttt{offset} < \text{cons}(p, f_a)$. Otherwise, the model behavior is undefined. The absolute token address is defined as $\texttt{address} = (\texttt{rd} + \texttt{offset}) \bmod \text{K}(p)$.

- The `tokens` method returns the number of tokens on the FIFO channel bound to an input port $p \in I$.

- The `commit` method increments the read pointer `rd` by the value of the `tokens` argument. In effect, this makes some more free places visible at the write interface of the FIFO channel. Note that this operation is performed by the RTE after an action function has been executed (cf. Algorithm 3.2).

As outlined above, channel generators create custom IP cores for the FIFO channels, if necessary. Complementary, port generators create concrete implementations of the abstract FIFO channel interface methods, depending on the binding of transactions to communication resources. Assume that an actor a is bound to a computing resource r_c, and a FIFO channel c connected to a is bound to a storage resource r_s. Then, the binding of a transaction to communication

		Channel	
	Binding	Memory	IP core
Actor	Processor	**Memory-mapped**	**Memory-mapped**
	IP core	**DMA**	**Signals**

Table 6.1: Depending on the binding of actors and channels to resources, typical token transport mechanisms can be identified.

resources[7] specifies how tokens are transported (or *routed*) from r_a to r_b (or vice versa), assuming that the transaction in question involves the actor a and the FIFO channel c. Obviously, it is not possible to describe every possible token transport scenario. However, considering only the resource types of r_a and r_b, three typical token transport mechanisms can be identified, which are summarized by Table 6.1. Thus, port generators usually have to create concrete implementations of the abstract FIFO channel interface methods based on these typical token transport mechanisms, which are discussed in more detail in the following.

6.2.1 Memory-Mapped Channel Access

If an actor a is bound to a processor, the FIFO channels connected to a are typically memory-mapped, and can therefore be accessed by means of pointers.

For channels bound to memory, the channel access methods created by the port generators are responsible for calculating the absolute addresses of requested tokens, and to update the read and write pointers associated with the FIFO channel. In this case, additional synchronization mechanisms may be required for inter-resource channels such that updates to the FIFO channel become visible to the involved resources in a consistent manner. For example, one must ensure that new token values become visible at the read interface *before* the value of the incremented write pointer becomes visible, as otherwise, old token values may be read. To this end, locks or (potentially more efficient) memory barriers can be used.

For channels bound to hardware, a more efficient memory-mapping scheme can be used in principle. In particular, it is not necessary to memory-map the read pointer, the write pointer, and the memory associated with the FIFO channel. Instead, the abstract channel access methods as summarized by Figure 6.2 can be directly implemented by the IP core created by the corresponding channel generator. In this case, the channel access methods created by the port generators are merely responsible for propagating the arguments and return values to and

[7] Note that in the general case, it is possible to bind a transaction to multiple communication resources, which is known as *multi-hop communication*.

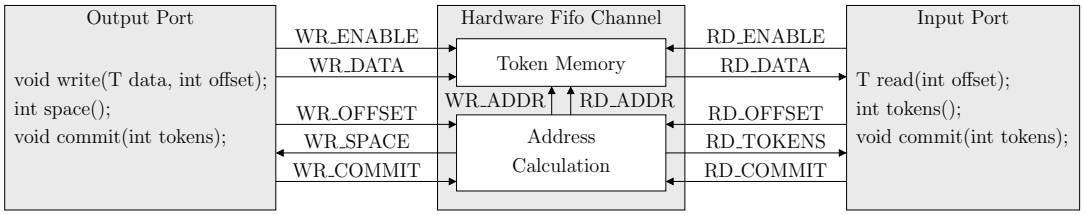


Figure 6.3: Hardware FIFO channel implementation.

from the memory-mapped registers of the channel IP core, while the token address calculation, as well as the updates to the read and write pointers are performed by the channel IP core. Note that a concrete implementation of a channel IP core is described in Section 6.2.3.

6.2.2 DMA-Based Channel Access

If an actor a is bound to hardware, and a FIFO channel connected to a is bound to memory, the channel access methods created by the port generators must issue memory read and write commands similar to an actor which is bound to a processor. To this end, the actor IP core created must be configured with the physical base address of the FIFO channel. This can be done either statically at compile time or dynamically at run time. In case of the latter, a memory-mapped register can be additionally allocated for the actor IP core which receives the base address of the FIFO channel. Note that if an operating system like Linux is used, the physical base address of the FIFO channel may not be known until run time, which therefore requires the FIFO channel to be configured dynamically. In the context of this thesis, DMA-based channel accesses have not been evaluated. Instead, the results focus on memory-mapped channels for the purpose of software synthesis, and on channels mapped to hardware for the purpose of hardware synthesis.

6.2.3 Signal-Based Channel Access

Finally, an actor a may be bound to hardware, and a FIFO channel connected to a may also be bound to hardware. In this case, the channel access methods created by the port generators must read or drive the input/output signals of the channel IP core created by the corresponding channel generator (cf. Figure 6.3). Note that the created channel access methods are at register-transfer level (RTL) level, and may require one or more clock cycles to complete. In the context of the proposed synthesis framework, SystemC code is generated.

The input/output signals of the channel IP core and their usage by the channel access methods created by the port generators can be summarized as follows:

- The number of tokens and free places is provided by the RD_TOKENS and WR_SPACE signals, respectively. These signals are read by the `tokens` and `space` methods.
- In order to write a token, the desired token value and offset must be provided via the WR_DATA and WR_OFFSET signals, and the WR_ENABLE signal must be asserted for one cycle. These signals are driven by the `write` method.
- In order to read a token, the desired offset must be provided via the RD_OFFSET signal, and the RD_ENABLE signal must be asserted for one cycle. The requested token is returned via the RD_DATA signal in the next clock cycle. These signals are driven and read by the `read` method.
- In order to make some more tokens visible, the WR_COMMIT signal must be set to the corresponding number. The channel's control logic samples this signal each clock cycle and advances the write pointer accordingly. This signal is driven by the `commit` method of the write interface.
- In order to make some more free places visible, the RD_COMMIT signal must be set to the corresponding number. The channel's control logic samples this signal each clock cycle and advances the read pointer accordingly. This signal is driven by the `commit` method of the read interface.

Finally, the channel generator is responsible for creating the channel IP core, which comprises the token memory and the address calculation logic. Note that it is assumed that the writing of a token takes exactly one cycle, while the reading of a token takes exactly two cycles. For Xilinx FPGAs, this assumption can be satisfied by binding the token memory to block RAM (BRAM). However, for variable latency memory accesses, an ACK signal could be implemented with little effort.

6.3 Computation Synthesis

As outlined above, communication synthesis is performed by port generators and channel generators. In this section, the generators pertaining to computation synthesis are discussed.

6.3.1 Software Synthesis

For actors bound to a processor, the generators pertaining to computation synthesis generate sequential C++ code which is further processed by a compiler in order to obtain an implementation at the instruction level (cf. Figure 6.1).

- *Actor generators* are responsible for synthesizing the guard functions and action functions that can be used by transitions. At a glance, this requires rewriting of the AST extracted from the input model. For example, port accesses must be replaced by the concrete channel access methods created by the corresponding port generators as described in Section 6.2. Analogously, task list evaluation and execution requests must be replaced by the concrete implementation created by the corresponding task list generators (described below).

- *Transition generators* are responsible for synthesizing the evaluation phase and execution phase of transitions according to Algorithms 3.1 and 3.2. Concerning the evaluation phase, code is generated which checks the availability of tokens and free places according to the values of cons and prod, and invokes the guard function of the transition. Note that the current actor mode is evaluated by task generators in this case (described next). Concerning the execution phase, code is generated which invokes the action function of the transition, and subsequently advances the read/write pointers of the involved FIFO channels. Note that the port generators provide the required channel access methods, and the actor generator provides the synthesized guard functions and action functions.

- *Task generators* are responsible for synthesizing the evaluation phase and execution phase of tasks according to Algorithms 5.1 and 5.3. Concerning the evaluation phase, actor modes can be synthesized into a C++ enumeration. Then, a switch-case statement over the current actor mode efficiently determines the subset of active transitions. Subsequently, the transitions in this set are evaluated in a round-robin fashion until an enabled transition has been found. The transition generators provide the required evaluation and execution methods of transitions.

- *Task list generators* are responsible for synthesizing the (binding-aware) evaluation phase and execution phase of task lists according to Algorithms 5.5 and 5.6. In particular, task list generators have to synthesize the distributed evaluation and execution of tasks. This encompasses two tasks (cf. Section 5.2): First, the evaluation and execution of tasks must be asynchronously started on resources. Second, a task must wait until its predecessor tasks are finished. Suitable shared data structures can be placed in shared memory or in memory-mapped registers, depending on the binding of child actors to resources. Some inter-processor synchronization mechanism can then be used to notify other processors that the shared data structures have been modified. Typical synchronization mechanisms are the polling of flags, inter-processor interrupts, or, if an operating system

is used, synchronization primitives offered by the operating system like *mutexes* and *condition variables* (Linux), or *events* (Windows).

As task lists are currently created during the elaboration phase, the computation of the topological ordering of tasks can be performed by the generators at compile time. In the future, this task could be postponed to run time in order to support the dynamic creation of task lists.

The functional simulation used in Section 5.4 in order to evaluate synthetic synchronous dataflow (SDF) graphs mostly corresponds to the software synthesis outlined in this section. The key difference is that the input model has not been translated from C# to C++. Synthesis results for the JPEG decoder from Figure 2.3 on page 12 are presented in Section 6.3.3.

In the next section, the hardware synthesis is described. It can be observed that the generators instantiated for the purpose of software synthesis largely correspond to the generators instantiated for the purpose of hardware synthesis. Thus, for generators discussed in this section, only the most important differences are discussed in the next section.

6.3.2 Hardware Synthesis

For actors bound to hardware, the generators pertaining to computation synthesis transform the untimed hierarchical input model into a *bus-cycle-accurate* SystemC representation, which is further processed by a (commercial) HLS tool in order to obtain an implementation at the logic level (cf. Figure 6.1). While sequential SystemC code can be generated in principle, certain optimizations based on the underlying dataflow model of computation can be performed in order to improve the performance of the synthesized model.

- *Actor generators* still generate sequential code for guard functions and action functions. In this case, the abstract channel access methods are typically replaced by cycle-accurate channel access methods which implement a certain input/output protocol (cf. Sections 6.2.2 and 6.2.3). Thus, the generated *bus-cycle-accurate* SystemC code is still untimed except for code blocks corresponding to channel accesses. Note that commercial HLS tools can be typically instructed to observe the user-defined scheduling of operations in such cycle-accurate code blocks. For example, the employed *Cynthesizer* [For14] HLS tool uses pragmas for this task. In order to reduce the latency of guard functions and action functions, tokens may be cached. This optimization is described in more detail in Section 6.3.3.

- *Transition generators* correspond to those instantiated for the purpose of software synthesis, as the availability of tokens and free places should still be checked before the guard function is evaluated.

- *Task generators* can exploit the fact that the evaluation of transitions is side-effect free and thus can be performed in parallel. The parallel evaluation of transitions is described in more detail in Section 6.3.3.

- *Task list generators* largely correspond to those instantiated for the purpose of software synthesis. However, child actors which are also bound to hardware are effectively inlined into the functional composite actor, resulting in a single SystemC module which is synthesized to a single IP core by subsequent synthesis tools. Note that independent tasks can be evaluated and executed in parallel in principle.

In summary, the generated bus-cycle-accurate SystemC model implements the channel accesses of guard functions and action functions in a cycle-accurate way, whereas the remaining behavior of an actor is still untimed. Bus-cycle accurate models may be used to explore the communication design space in order to decide how the abstract FIFO channels should be bound to resources available in the platform (e.g., shared memory vs. custom IP cores).

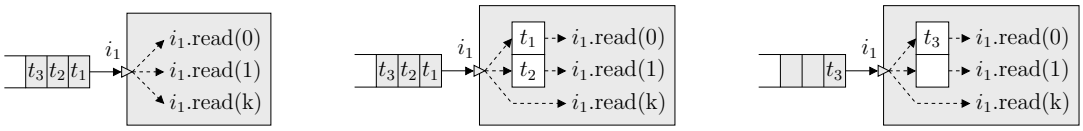
6.3.3 Token Caching and Parallel Evaluation of Transitions

In this section, the optimizations outlined in the previous section are described in more detail. In particular, the caching of tokens and the parallel evaluation of transitions is discussed. These micro-architectural optimizations have been presented in [ZHFT12b].

Token Caching

The token access as described in Section 6.2 requires the designer to provide an offset for each token to be read. While this permits random access to the tokens in the channel, it also sequentializes token accesses, as only one token can be read each clock cycle. Thus, if a guard function or action function accesses multiple tokens (possibly multiple times), latency may increase. For example, the three port accesses in Figure 6.4a read tokens from the same channel via input port i_1. Assuming that a read operation has a latency of two clock cycles, and that a new read operation can be started each clock cycle, the three port accesses take four clock cycles to complete.

In order to hide this token access latency, a token cache can be added to the read interface (cf. Figure 6.4b). Note that the token cache only uses the abstract channel interface methods as shown in Figure 6.2. Thus, the proposed token cache can be used independently from the concrete implementation of the interface methods provided by the port generator which is associated with the actor input port in question.



a) No token cache

b) Token cache before commit

c) Token cache after commit

Figure 6.4: Token access with a token cache.

In order to prevent the generation of large multiplexer structures, the cache module is only used for token accesses with a constant offset. For token accesses with a variable offset, the random-access protocol is still used, and thus, the cache module also has to provide this protocol. In summary, the cache module works as follow: If a token access with a variable offset is active, it is simply forwarded by the cache module to the `read` method provided by the port generator. If no such access is active and the cache is not yet full, the cache module itself invokes the `read` method provided by the port generator and stores the returned token in the cache.

In order to prevent a function from reading invalid token values from the cache, the number of available tokens as reported by the cache module is calculated as follows: If the cache is not yet full, the number of cached tokens is returned. If the cache is full, the actual number of tokens available in the FIFO channel is returned by invoking the `tokens` method provided by the port generator. Note that this behavior does not introduce deadlocks into the model.

For example, consider Figure 6.4b, where a cache module with space for two tokens has been inserted. Assuming that the cache is full, the two port accesses with a constant offset can concurrently access tokens t_1 and t_2, while the port access with a variable offset k can randomly access token t_1, t_2 or t_3. Thus, in this case, only two clock cycles are required to perform all three port accesses.

As described in Section 6.2, the read and write pointers of a FIFO channel are incremented during the commit phase which is performed after the execution of an action (cf. Figure 6.4c). Thus, token values do not change during the execution of a guard function or action function, assuming point-to-point FIFO channels as specified by the underlying dataflow model of computation. Thus, the cache is cleared only during the commit phase, and is subsequently refilled (cf. Figure 6.4c).

In order to access the cached tokens, the cache module provides a `readCache` method which simply returns the value of the register corresponding to the specified constant token offset. Finally, whether or not a token offset is constant (i.e., an *integer constant expression*) is determined while traversing the AST of the function body.

Parallel Evaluation of Transitions

Task generators instantiated for the purpose of software synthesis evaluate the transitions of the associated task sequentially in a round-robin fashion. While this is a sensible approach for software synthesis, hardware synthesis should exploit the fact that the evaluation of transitions is side-effect free and thus can be performed in parallel. Note that in order to achieve a concurrent RTL implementation after HLS, transitions have to be evaluated in concurrent SystemC processes. However, evaluating transitions in concurrent SystemC processes raises some problems, as discussed next.

First, the channel interface methods provided by port generators may now be invoked by different processes. This may result in concurrent token accesses which must be arbitrated appropriately. To this end, an arbitration module is provided which arbitrates concurrent token accesses from different processes. Similar to the token cache module, the arbitration module only uses the abstract channel interface methods as shown in Figure 6.2. Thus, the proposed arbitration module can be used independently from the concrete implementation of the interface methods provided by the port generator which is associated with the actor input port in question. Note that token accesses may take more than two cycles due to the arbitration of concurrent token accesses. However, token requests from different processes can be served simultaneously if they refer to the same token (i.e., if they specify the same token offset). Moreover, the arbitration mechanism is only required for variable token offsets, as for constant token offsets the token is fetched from the token cache without driving any signals (assuming that a token cache module has been instantiated for the port in question).

Second, transferring the evaluation of transitions into separate processes requires the introduction of enable and ready signals. The enable signal can be implemented as a reset signal which is active until the transition should be evaluated. The ready signal is asserted when the transition has been evaluated, and stays active until the transition is re-evaluated. Thus, the process which invokes the action functions can check the ready signals of the various evaluation processes at any time. It remains the question when to reset an evaluation process. Due to the underlying model of computation, the evaluation result only depends on the actor state and tokens on incoming channels. Thus, a conservative approach is to reset all evaluation processes after an action function has been executed, and the corresponding tokens have been consumed.

Results

In order to show the applicability of the proposed software synthesis, hardware synthesis, and optimizations, the JPEG decoder depicted in Figure 2.3 on page 12 has been synthesized. Concerning the software test case (SW), all actors of the

Synthesis	Latency	Throughput	LUTs	Registers	DSP48s
SW	60.8ms	≈ 40 FPS	-	-	-
HW-Seq	28.3ms	≈ 40 FPS	13692	9293	8
HW-Tok	17.8ms	≈ 60 FPS	13389	11483	9
HW-Par	11.5ms	≈ 100 FPS	11692	10285	9

a) End-to-end performance and resource utilization. SW: Software synthesis (Intel Xeon E7-8837). HW-Seq: Sequential evaluation of transitions, no token cache. HW-Tok: Sequential evaluation of transitions, token cache. HW-Par: Parallel evaluation of transitions, token cache.

Actor	Synthesis	Latency	LUTs	Registers	DSP48s
HuffmanDecoder	HW-Seq	14.0ms	8203	5200	0
	HW-Tok	9.3ms	7604	4806	0
	HW-Par	9.5ms	6590	3851	0
InverseZRL	HW-Seq	12.1ms	685	302	0
	HW-Tok	7.3ms	566	309	0
	HW-Par	7.7ms	369	131	0
InverseQuant	HW-Seq	**25.3ms**	926	294	1
	HW-Tok	**15.6ms**	800	274	1
	HW-Par	7.7ms	374	244	1
InverseZigZag	HW-Seq	12.2	133	82	0
	HW-Tok	11.9	134	81	0
	HW-Par	**9.7**	78	46	0
IDCT2D	HW-Seq	11.2ms	3637	3286	6
	HW-Tok	6.9ms	4060	5887	8
	HW-Par	7.4ms	4060	5887	8
YCbCr2RGB	HW-Seq	12.7ms	108	129	1
	HW-Tok	7.7ms	225	126	0
	HW-Par	8.5ms	221	126	0

b) Per-actor latencies and resource utilization for the hardware synthesis.

Table 6.2: a) End-to-end performance and resource utilization of the synthesized JPEG decoder from Figure 2.3 on page 12. b) Latencies and resource utilization of selected individual actors. The latencies corresponding to the bottlenecks have been highlighted for each test case.

model have been automatically translated to C++ as described in Section 6.3.1. The resulting code has been compiled and executed on an Intel Xeon E7-8837 running at 2.67GHz. Concerning the hardware test cases (HW-Seq, HW-Tok, and HW-Par), all actors have first been automatically translated into a bus-cycle-accurate SystemC representation as described in Section 6.3.2. Subsequently, all actors except the `JPEGSource` and `ImageSink` actors have been synthesized

to RTL by the ForteDS Cynthesizer HLS tool [For14]. For this purpose, a Xilinx Virtex-5 XC5VLX110T FPGA has been chosen as target platform. The target clock period has been set to 10ns. Synopsys Synplify Pro has been used for logic synthesis, and the Xilinx tools have been used for place and route. Note that the `JPEGSource` and `ImageSink` actors represent the testbench and have not been further processed.

The overall results are shown in Table 6.2a, while the results for some individual actors are shown in Table 6.2b. In the general case, the given latencies correspond to the time required to decode a single JPEG picture with QCIF resolution (i.e., 176x144 pixels). In case of the SW test case, the latency corresponds to the average time required to decode the same picture five times. In case of the HW test cases, it should be noted that the JPEG decoder is pipelined based on minimum coded units (MCUs), but the given latencies of individual actors also refer to the decoding of a whole picture, which consists of 396 MCUs.

Concerning the SW test case, the peak throughput of approx. 40 FPS can be achieved by decoding Motion-JPEG videos with at least 200 frames (with the same QCIF resolution). Compared to the latency of a single picture, the micro-architectural optimizations like instruction pipelining and caches of the Intel Xeon processor considerably increase the throughput of the JPEG decoder synthesized to software.

Concerning the HW-Seq test case where transitions are evaluated sequentially and no token cache is used, the peak throughput of approx. 40 FPS is limited by the slowest actor in the actor chain, which is the `InverseQuant` actor in this case. It should be noted that the throughput is comparable to the throughput achieved by the SW test case. However, the Intel Xeon processor is running at 2.67GHz, while the IP core corresponding to the JPEG decoder synthesized to hardware is running at only 100MHz.

It can be observed that the latency is improved significantly by the presented optimizations: Token caching (test case HW-Tok) reduces the latency by approx. 40% w.r.t. the HW-Seq test case, while token caching combined with the parallel evaluation of transitions (test case HW-Par) reduces the latency by approx. 60% w.r.t. the HW-Seq test case. At a glance, the largest latency savings can be observed in modules with a majority of token accesses with constant offsets and many transitions. On the other hand, actors with a majority of token accesses with variable offsets and less transitions do not contribute much to the latency improvements. According to the bottleneck actors of the corresponding test cases, the throughput could be increased to approx. 60 frames per second and 100 frames per second, respectively. However, it should be noted that the QCIF resolution is quite small. For a typical high-definition resolution of 1280x720 pixels, the throughput can be expected to decline to approx. 3 frames per second, which indicates that the input model and the hardware synthesis should be further optimized in future work. For example, micro-architectural optimizations like a

pipelined or speculative evaluation/execution of transitions could be considered. In particular, the latter can exploit the fact that only those transitions may become active which are attached to the possible target modes of the transition which is currently being executed.

Concerning the LUT utilization, a reduction can be observed when applying the presented optimizations. For the token caching alone, the slight reduction probably stems from the fact that more tokens are accessed via constant offsets than via variable offsets. In this case, less logic is needed to implement the random-access protocol for token accesses, due to the fact that a majority of token accesses is directly wired to the registers of the token cache module. In case of the parallel evaluation of transitions, the LUT utilization is reduced even more due to the fact that guard functions are no longer inlined (and possibly duplicated) into the main process.

As could be expected, the register utilization in case of the sequential evaluation of transitions with token caching increases (by approx. 23%) compared to the baseline synthesis due to the additional registers used by the token cache. However, this increase in register usage is alleviated by the parallel evaluation of transitions, which reduces the number of registers used for implementing the guard functions, due to the same reason as described for the LUTs.

Finally, the optimized variants require nine DSP48 resources instead of eight required by the baseline synthesis. Concerning the parallel evaluation of transitions, this may be caused by less possibilities for resource sharing when functionality is split into separate SystemC processes, as HLS tools typically do not perform resource sharing across process boundaries.

6.3.4 Inter-Process Resource Sharing

The hardware synthesis as described in the previous sections leaves all resource sharing decisions to the HLS tool which is used to further process the generated bus-cycle-accurate SystemC model. While in this case, all logic is inlined into a single SystemC process which maximizes the resource sharing possibilities, it is difficult to express parallelism using this approach. The parallel evaluation of transitions partially solved this problem by generating concurrent SystemC processes for the transitions. While this approach reduces resource sharing possibilities for the subsequent HLS of the bus-cycle-accurate model, even more resources could possibly be saved by this approach if some transitions specify the same guard function. In this case, only one instance of the guard function is synthesized instead of synthesizing all inlined guard function calls. The parallel evaluation of transitions may also result in improved performance as shown in the last section.

For a single actor, guard functions that are used by more than one transition may not be encountered very often. While the parallel evaluation of transitions

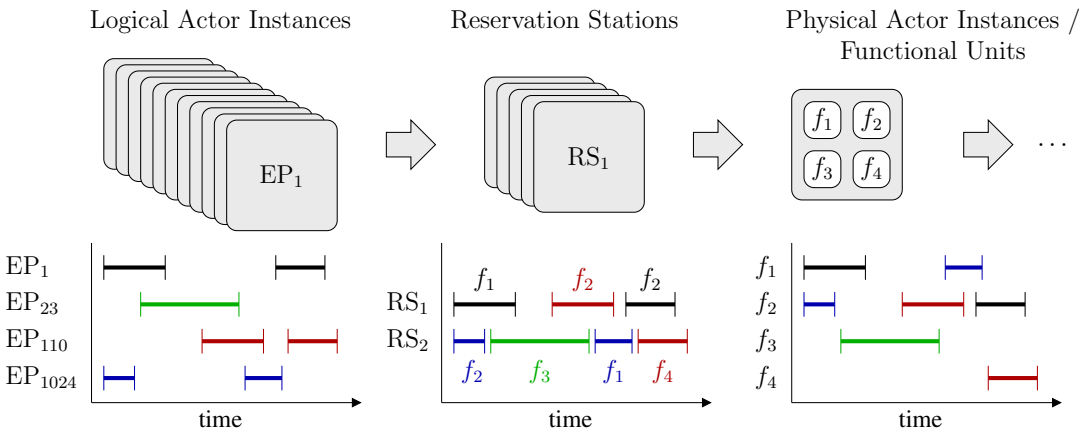


Figure 6.5: Overview of the proposed inter-process resource sharing approach: A large number of logical actor instances is bound to a smaller number of reservation stations, which are in turn bound to an even smaller number of physical actor instances (which are comprised of functional units). In turn, this creates opportunities for resource sharing between functional units.

may still improve performance, resources cannot be shared in this case. However, it can be observed that for a wide range of applications, the same *actor* is instantiated more than once. In particular, multimedia applications can benefit from modeling the processing of different audio/video components as a (hierarchical) actor which is instantiated according to the number of components.

Example 6.1. The composite actor shown in Figure 5.2 on page 93 can be instantiated three times in order to model the parallel decoding of the image components (Y, Cb, Cr) of a JPEG image. In this case, each of the child actors `InverseQuant`, `InverseZigZag`, and `IDCT2D` is instantiated three times.

In such a scenario, it is possible to realize resource sharing across process boundaries not only based on guard functions, but also to consider action functions. For a small number of identical actors, this inter-process resource sharing approach has been proposed in [ZHFT12a].

Example 6.2. Another application domain where the proposed resource sharing approach can be employed is the networking domain. Here, the *transport layer* according to the Open Systems Interconnection (OSI) reference model [DZ83] is responsible for providing end-to-end communication services for applications. Typically, the complex behavior of a single communication endpoint (EP) depends on packets received from the peer EP, or on data received

from the application. Here, the proposed dataflow model can be used to efficiently implement the behavior of a single communication EP as shown in [ZFH+10].

In such a scenario, a very large number of identical actors must be accommodated. In this case, the inter-process resource sharing approach as presented in [ZHFT12a] must be adapted as described in [ZHF+14].

Figure 6.5 summarizes the proposed inter-process resource sharing approach: If a very large number of identical actors is to be processed, they are first bound to a smaller number of *reservation stations*. Subsequently, the reservation stations are bound to an even smaller number of *physical actor instances*. Each physical actor instance is comprised of *functional units*, which in the proposed dataflow model simply correspond to the synthesized guard functions and action functions. In turn, this creates opportunities for resource sharing between functional units, which is, however, not considered in this thesis.

The logical actor instances are implemented by means of the (hierarchical) dataflow model presented in Sections 3.1 and 5.1. Before the input model expected by the resource sharing approach is described in more detail, the example which is used throughout the remaining chapter is outlined in the next section. Subsequently, the inter-process resource sharing approach is described.

System-Level Overview

Example 6.3. The example used in the following is based on an InfiniBand (IB) [Inf14] network adapter contained in a PCI Express (PCIe) system (cf. Figure 6.6). PCIe employs point-to-point links to overcome the limitations of a shared bus. In a PCIe system, a *root complex device* connects the processor and memory subsystem to the PCIe switch fabric comprised of one or more *switch devices*. As a packet-based protocol, PCIe consists of three layers as known from the networking domain, namely the *transport layer*, the *data link layer*, and the *physical layer*.

The network adapter implements the InfiniBand *SEND* operation defined on the transport layer. With a SEND operation, the local EP transmits data to a remote EP. To this end, each EP has an associated send queue (SQ) and receive queue (RQ) where work queue elements (WQEs) are inserted by the user. A WQE ("wookie") specifies where to fetch or store the transmitted data. Each EP processes its posted WQEs, and, for each finished WQE, places a completion queue element (CQE) ("cookie") in an associated completion queue (CQ) polled by the user. The InfiniBand specification provides for a total of 2^{24} EPs. Note that an EP can simultaneously process WQEs from its SQ and its RQ. The SQs, RQs, and CQs are stored in main memory (cf. Figure 6.6).

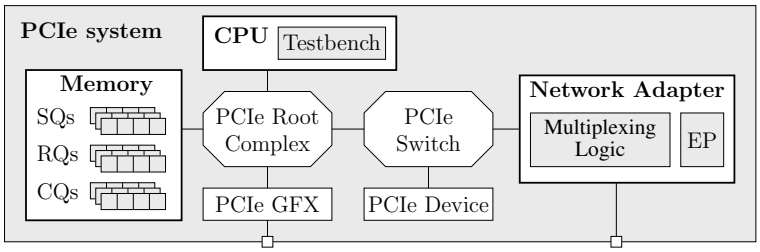


Figure 6.6: System-level overview of the running example based on an InfiniBand network adapter in a PCIe system.

In order to reduce the complexity of the example, only the *unreliable connection* mode specified by InfiniBand has been implemented. In this mode, the requester receives no acknowledgments for transmitted packets, and no guarantees concerning the packet order are given.

The behavior of a single EP is summarized by the sequence diagram shown in Figure 6.7. In order to reduce the modeling complexity, the SQ and RQ processing logic is implemented by means of hierarchical modes as described in Section 3.2. To be more precise, an EP consists of a top-level AND mode, whose XOR child modes implement, amongst others, the SQ and RQ processing logic. An additional XOR child mode (which is not reflected by Figure 6.7) handles memory-mapped configuration register accesses by applications (or the driver if an operating system like Linux is used).

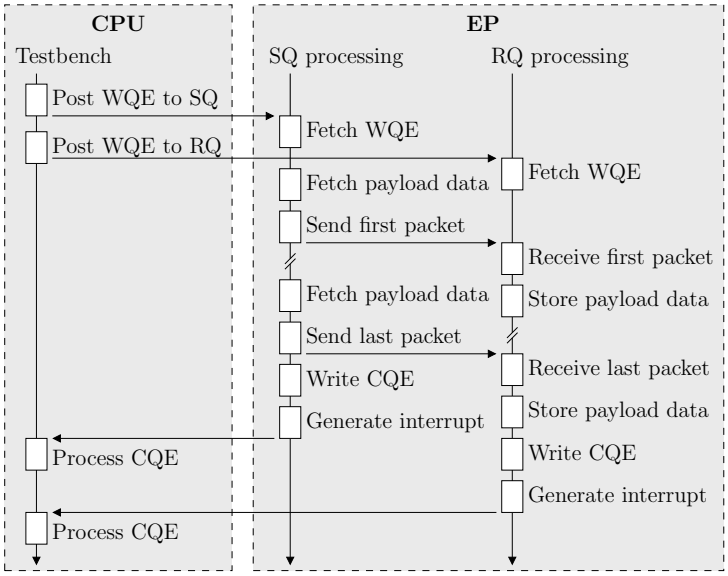


Figure 6.7: Work queue element (WQE) processing by an EP.

Input Model and Modeling Extensions

The input model expected by the inter-process resource sharing approach consists of a structural composite actor γ which has n instances of identical functional composite child actors a_1–a_n. In the following, these child actors are called *logical actor instances*.

Tokens from an input port p of the composite actor γ (or from an output port p of a child actor of γ) may have to be forwarded to an input port p' of a logical actor instance. In principle, this could be achieved by a dispatch actor with one input port i_1 and n output ports o_1–o_n. Then, a port-to-port binding (or a channel) (p, i_1) is added, and a channel $(o_k, a_k.p')$ for each logical actor instance $1 \leq k \leq n$. Analogously, tokens produced by a logical actor instance on an output port p' typically may have to be forwarded to an output port p of the composite actor γ (or an input port p of a child actor of γ). In this case, a merge actor with n input ports i_1–i_n and one output port o_1 could be used. Then, a channel $(a_k.p', i_k)$ for each logical actor instance $1 \leq k \leq n$ is added, and a port-to-port binding (or a channel) (o_1, p).

While these user-defined dispatch and merge actors would not require any modeling extensions, this approach has some problems: The number of transitions of these actors corresponds to the number of logical actor instances, which may become very large. This is due to the fact that these actors explicitly encode the target and source logical actor instances by means of dedicated output and input ports. However, these user-defined dispatch and merge actors cannot be automatically transformed to support a different token forwarding scheme based on physical actor instances instead of logical actor instances in the general case. Thus, if the logical actor instances are bound to fewer physical actor instances, additional multiplexing logic would be required to forward tokens from the output ports of the dispatch actor to the input ports of the few physical actor instances, while additional demultiplexing logic would be required to forward tokens from the output ports of the few physical actor instances to the input ports of the merge actor.

Concerning the user-defined dispatch actor, a more efficient approach decouples the decision to *which* logical actor instances tokens should be forwarded from the concrete implementation of the forwarding mechanism. To this end, an additional communication primitive is provided which can be used to replace the user-defined dispatch actor [ZFH+10]: In order to forward a token t to the appropriate logical actor instance a_k, we only have to know the logical actor instance associated with t. This task can be performed in isolation by an actor which transforms a token t into a token (t, k) which contains the original token t and additionally specifies the index of the logical actor instance a_k to which t should be forwarded. This process is referred to as *token coloring* in the following, and the compound token (t, k) is referred to as *colored token*.

The colored token (t, k) is produced on a *dispatch channel* which basically behaves like the dispatch actor described above: The original token t of a colored token (t, k) stored on the dispatch channel is made available at the attached input port p' of the corresponding logical actor instance a_k. However, in contrast to user-defined dispatch actors, dispatch channels are implemented by the generators of the proposed synthesis framework, and can therefore be easily adapted in order to support physical actor instances instead of logical actor instances. Dispatch channels are $1 : n$ connections, i.e., they connect exactly one output port with n input ports, possibly via port-to-port bindings.

The size of a dispatch channel refers to the number of colored tokens which can be stored on the dispatch channel. In turn, tokens are assumed to be consumed in a sequential manner, i.e., if the sequence of colored tokens stored on a dispatch channel is $S = \langle (t_1, k_1), (t_2, k_2), \ldots, (t_n, k_n) \rangle$, only the first j tokens of the same color are visible to the logical actor instance a_{k_1}, i.e., $j = \max_i \{\forall q, 1 \leq q \leq i : k_1 = k_q\}$. The designer has to consider these semantics: In particular, deadlocks could be introduced into the model if the logical actor instance a_{k_1} does not eventually consume the first j tokens, because in this case, subsequent colored tokens never become visible at their respective logical actor instance. Note that colored tokens in dispatch channels always target a specific logical actor instance. Thus, if only dispatch channels are used alongside regular point-to-point FIFO channels, the resulting input model is still conflict-free, even though dispatch channels are not point-to-point connections in the general case.

In order to replace the user-defined merge actors, *merge channels* are introduced. Merge channels are $n : 1$ connections, i.e., they connect n output ports with exactly one input port p, possibly via port-to-port bindings. In this case, a token produced by a logical actor instance a_k on an output port p' is made available at the only connected input port p as a colored token (t, k). Here, the question arises in which order the merge channel should forward tokens from different logical actor instances to the connected input port p. If no restrictions can be imposed on the forwarding of tokens from different logical actor instances, the multiplexed tokens may appear in the wrong order at p, and may have to be re-ordered by an intermediate actor. Unfortunately, in contrast to the actors which perform the coloring of tokens for dispatch channels, these token re-ordering actors would be more complex. In particular, the re-ordering of tokens requires the buffering of tokens.

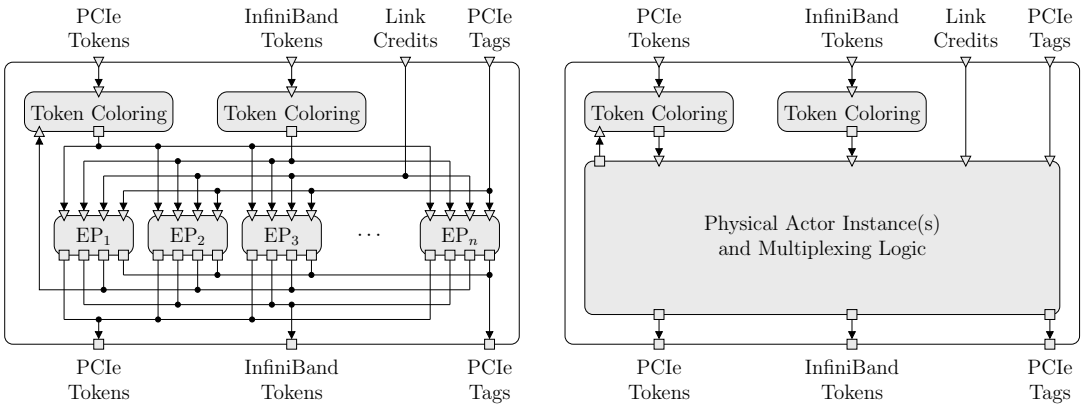
A possible solution is to extend the logical actor instances to produce tokens (t, l) where $l \in \{\top, \bot\}$ is a flag that indicates whether the token t is the last token in a sequence of tokens that must not be interleaved with tokens from other logical actor instances. Then, the semantics of merge channels are as follows: Tokens (t, l) produced by a logical actor instance a_k on an output port p' are made available at the only connected input port p as a colored token (t, k). However, only after a token with $l = \top$ has been forwarded, tokens from a different logical actor instance can be forwarded.

The size of a merge channel refers to the number of colored tokens which can be stored on the merge channel. Thus, in principle, conflicts may be introduced into the resulting input model if merge channels are used. An implementation must resolve these conflicts appropriately, e.g., by allocating dedicated FIFO buffers such that the logical actor instances can produce tokens in isolation before the tokens are merged onto a single channel.

In real-world designs, it is typically desirable to model limited resources. In principle, this could be achieved by a dedicated *server actor* which manages the resources in question. Then, actors which want to acquire a resource can send a request token to this server actor. When a resource becomes available, the server actor sends an acknowledgment token to the next requesting actor. When the actor has finished using the resource, it sends a release token to the server actor.

While feasible, this approach results in more complex actor finite state machines (FSMs) which must model this server/client behavior. In contrast, a set of limited resources can also be modeled by a simple semaphore, which is basically a counter that is decremented when a resource is acquired, and incremented when a resource is released. This behavior is provided by *semaphore channels*: In contrast to dispatch channels, tokens in a semaphore channel are not associated with a specific logical actor instance, and thus, can be consumed by any logical actor instance. In turn, these tokens are referred to as *colorless tokens* in the following. Semaphore channels are $n : m$ connections, i.e., n output ports are connected to m input ports, possibly via port-to-port-bindings. Thus, conflicts between actors may occur as semaphore channels are not point-to-point channels in the general case, and colorless tokens may be consumed by any logical actor instance. An implementation must resolve these conflicts appropriately. However, for semaphore channels, it can be assumed that always enough free places are available in order to produce some tokens. This is due to the fact that a correct model must never produce more tokens on a semaphore channel than previously consumed from the same channel, i.e., resources must have been acquired by an actor before they are released.

In the following, it is assumed that for a given dispatch channel or semaphore channel which is connected to the input ports of the logical actor instances contained in the structural composite actor γ, no input ports of actors not contained in γ are additionally connected. Analogously, it is assumed that for a given merge channel or semaphore channel which is connected to output ports of the logical actor instances contained in the structural composite actor γ, no output ports of actors not contained in γ are additionally connected. These restrictions ensure that the additional communication primitives can be locally replaced by the regular point-to-point FIFO channels (cf. Definition 5.1) by model transformations. Moreover, conflicts between transitions of logical actor instances induced by merge channels or semaphore channels can therefore be resolved in a local manner.



a) Network adapter modeled by concurrent actor instances

b) Network adapter with multiplexed actor instances

Figure 6.8: The proposed inter-process resource sharing approach transforms the input model (a) into a model where the logical actor instances are multiplexed onto a few physical actor instances (b)

Example 6.4. The input model of the InfiniBand network adapter is shown in Figure 6.8a. The four input ports and three output ports correspond to tokens from/to the PCIe switch fabric, tokens from/to the link interface, tags which are assigned to memory read requests issued by the model, and link credits consumed by the model before sending a token to the link interface. Note that link credits are generated by the link interface, and not by the logical actor instances EP_1–EP_n.

Link credits and PCIe tags are colorless tokens and can therefore be consumed by any EP. In contrast, PCIe and IB tokens are colored tokens, as they always target a specific EP. Note that for a data path width of 32 bits, PCIe and IB packets are transmitted by means of multiple 32 bit tokens. Thus, the actors which perform the token coloring must analyze the token stream accordingly. Typically, the target EP can be determined from the header tokens which are possibly followed by payload tokens.

In case of InfiniBand tokens, the resulting FSM is outlined in Figure 6.9. Note that the target EP is contained in the second token of the base transport header (BTH). Thus, the three preceding header tokens can only be produced onto the dispatch channel when the target EP is known, which requires some additional buffering of tokens. However, the concrete implementation is left to the designer. Considering Figure 6.9, three IB tokens have been assigned to $\mathrm{EP}_{\mathrm{green}}$, while the subsequent token has been assigned to $\mathrm{EP}_{\mathrm{red}}$.

Concerning PCIe tokens, the actor which performs the token coloring is slightly more complex. In case of write and read requests, the target EP is determined from the target address specified by a PCIe header token. In case

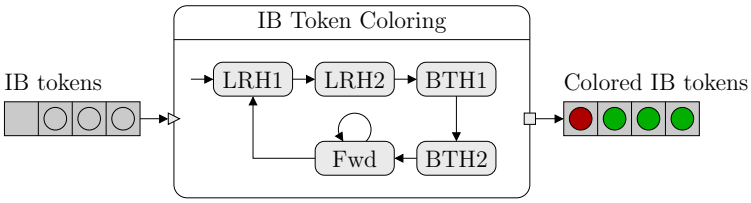


Figure 6.9: Token coloring for InfiniBand packets. The modes correspond to the tokens which make up the local route header (LRH) and BTH of IB packets. Note that depending on the LRH, a global route header (GRH) may be inserted between the LRH and the BTH which is not reflected by the FSM but should be considered in the general case.

of a read response, the target EP is determined based on the tag which is contained in the read response. Note that the tag is contained in the read request issued earlier by an EP, and is simply copied into the read response by the target device in the PCIe switch fabric which processes the read request. Thus, the PCIe token coloring actor must know which EP is associated with a given tag. For example, when sending a PCIe read request, an EP can also produce a token $(\text{tag}, \top)$ on an additional merge channel connected to the PCIe token coloring actor (cf. Figure 6.8a). In this case, the PCIe token coloring actor receives a token (tag, k), and can update its internal tag table accordingly. However, the concrete implementation is again left to the designer.

In principle, the result of multiplexing the logical actor instances onto the physical actor instances can be represented by a functional composite actor (cf. Figure 6.8b). Note that in this transformed model, only point-to-point channels are used. This functional composite actor can be generated, e.g., in order to perform functional verification of the multiplexing logic. In principle, this functional composite actor could also be used to perform hardware synthesis. However, the input model is chosen such that the generators employed by the synthesis framework can directly output the synthesized model without having to generate the functional composite actor. This enables optimizations during synthesis which cannot be represented in the proposed dataflow model, like the asynchronous execution of child actors. Note that such execution semantics of child actors are supported by the FunState model, which has been reviewed in Section 3.4.

Mapping of Logical Actor Instances to Reservation Stations

The first step of the inter-process resource sharing approach dynamically maps the logical actor instances a_1–a_n to a smaller set of *reservation stations* RS_1–RS_q,

$q \leq n$ (cf. [ZHF+14]). To this end, the following problems must be solved: First, the logical actor instances to be bound to reservation stations must be *selected* from the set of all logical actor instances. Second, the selected logical actor instances must be *scheduled* if not enough reservation stations are available. Third, the scheduled logical actor instances must be *bound* to reservation stations. Please note that in this case, the scheduling is performed prior to the binding of logical actor instances to reservation stations. This is made possible by the fact that each reservation station (RS) can execute any logical actor instance. In turn, the utilization of reservation stations is improved.

Concerning the selection of logical actor instances, a simple round-robin scheme is infeasible if it is to be expected that only a few logical actor instances have some enabled transitions. However, as described in Section 3.1, the availability of sufficient tokens and free places on channels is a necessary condition for a transition to be enabled. Therefore, logical actor instance can be selected based solely on tokens available on input ports, and free places available on output ports. In the following, it is assumed that logical actor instances are initially blocked on colored tokens. If this is not the case, all logical actor instances must be initially considered as potentially having enabled transitions. In principle, the proposed approach also supports such a scenario, as will be seen later.

The scheduling of selected logical actor instances is based on credits. However, this is explained in more detail after the binding of logical actor instances to reservation stations has been described, which can be outlined as follows: Given a colored token (t, k) from a dispatch channel, we first check whether the logical actor instance a_k is already bound to a reservation station RS_i. If this is the case, the original token t is extracted from the colored token, and forwarded to the corresponding reservation station. Otherwise, an *idle* reservation station is selected. An RS is idle if either no logical actor instance is bound to the RS, or all of the following conditions are satisfied:

- The input channels to the RS corresponding to dispatch channels must be empty. This requirement prevents the RS from consuming tokens which target a logical actor instance that is no longer bound to this RS. Note that this requirement implies the assumption that the RS eventually consumes all tokens from input channels. However, according to the sequential semantics of dispatch channels, this assumption must be satisfied anyway in order to not introduce deadlocks into the input model.

- The output channels from the RS corresponding to merge channels must either be empty, or the last token (t, l) produced onto a merge channel must correspond to the last token of a token sequence, i.e., $l = \top$. This requirement prevents the generation of spurious token sequences, because according to the semantics of merge channels, tokens must be forwarded from the same logical actor instance until a token $(t, \top)$ is encountered.

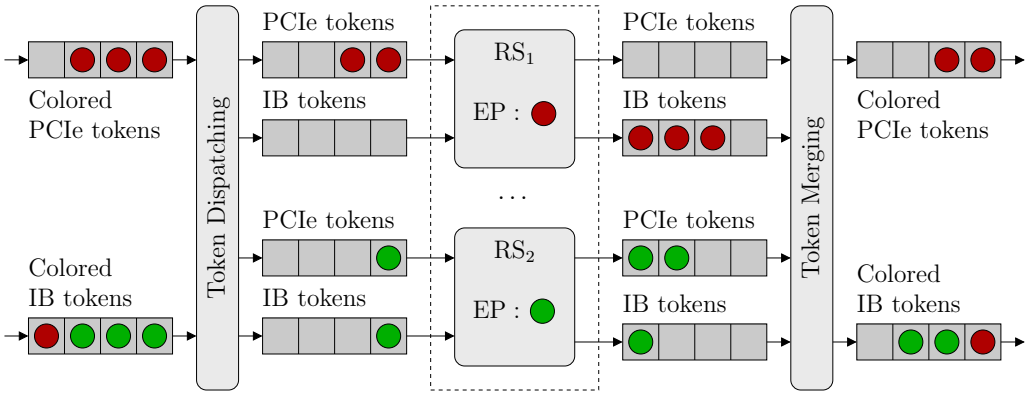


Figure 6.10: Data path of the InfiniBand example. For the sake of clarity, only dispatch and merge channels are shown. The dotted rectangle indicates the mapping of the reservation stations to the physical actor instances (cf. Figure 6.11).

Note that this requirement does not introduce deadlocks into the model, as the input model must already adhere to the semantics of merge channels.

- The RS must not currently evaluate or execute a transition. In particular, this requirement ensures that no action function of the RS is being executed, and that the logical actor instance currently bound to the RS therefore is in a consistent state.
- The RS should not have any enabled transitions. In this case, it is typically better to execute an enabled transition before evicting the logical actor instance currently bound to the RS. However, if a logical actor instance never blocks on missing tokens or free places, it may have to be evicted despite having enabled transitions. In the following, it is assumed that this scenario does not occur. Note that this is basically the same assumption as Requirement 4.2 on page 60 postulated for the actor classification.

Example 6.5. Considering Figure 6.10, $\mathrm{EP}_{\mathrm{red}}$ is currently bound to RS_1, while $\mathrm{EP}_{\mathrm{green}}$ is currently bound to RS_2. Both reservation stations are not idle, as there are some tokens on the input channels to each RS. The colored tokens are subsequently forwarded to the corresponding RS.

In principle, the selection of an idle reservation station to which the logical actor instance a_k should be bound to can be based on existing cache replacement strategies, like least recently used (LRU). A more informed selection strategy may inspect the first colored tokens on the dispatch channels and refrain from

selecting an RS for which colored tokens are queued. As implemented for the example, a simple *random selection* strategy is used, i.e., any idle RS is selected.

After an idle reservation station RS_i has been selected, the logical actor instance a_j currently bound to RS_i is *evicted* from RS_i. To this end, the actor state (cf. Definition 3.2) of a_j is written into some backing store (e.g., off-chip memory). Afterward, the logical actor instance a_k is assigned to RS_i. When the actor state of a_k has been fetched from the backing store, transitions of RS_i can be evaluated and executed.

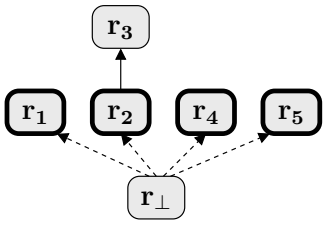
Assuming that the evicted logical actor instance is not solely blocked on colored tokens, but misses some colorless tokens or free places on output channels, it will never again be evaluated if no more colored tokens for the evicted logical actor instance are received. Thus, the evicted logical actor instance must be remembered in order to provide for an efficient selection when enough resources are eventually available (remember that a simple round-robin scheme is infeasible if it is to be expected that only a few logical actor instances have some enabled transitions). To this end, one or more *initiative queues* are allocated, where *initiative tokens* are enqueued. An initiative token consists only of the index of the logical actor instance which has been evicted. Thus, it is basically a colored token without data.

An initiative queue is associated with a set of requirements which must be met in order to dequeue an initiative token. A requirement may either encode the number of colorless tokens required on a semaphore channel, or the number of free places required on a merge channel. Remember that it is assumed that always enough free places are available on semaphore channels. The requirement sets are statically determined by analyzing the transitions of a logical actor instance. Note that transitions which consume at least one colored token are ignored, because in this case the actor is activated by the colored tokens as described above.

Example 6.6. Analyzing the transitions of an InfiniBand EP results in the requirement sets $\mathbf{r_1}$–$\mathbf{r_5}$ shown in Table 6.3a. For example, $\mathbf{r_1}$ specifies that one

Requirement Set	i_{tags}	i_{credits}	o_{pcie}	o_{ib}
$\mathbf{r_1}$	1	0	3	0
$\mathbf{r_2}$	0	0	4	0
$\mathbf{r_3}$	0	0	7	0
$\mathbf{r_4}$	0	6	0	6
$\mathbf{r_5}$	0	69	0	0

a) Requirement sets



b) Partial order

Table 6.3: Requirement sets computed from the transitions of an InfiniBand EP.

colorless tag token is required, in addition to three free places on the PCIe merge channel. Note that $\mathbf{r_1}$ corresponds to a PCIe read request issued by an EP.

In principle, an initiative queue could be allocated for each requirement set. While this approach retains the exact requirements, it may also lead to the allocation of many initiative queues. It can be observed that the requirement sets form a partial order under vector comparison "$<$". Given two resource requirements $\mathbf{r_1}$ and $\mathbf{r_2}$, $\mathbf{r_1} < \mathbf{r_2}$ if $\mathbf{r_1} \leq \mathbf{r_2}$ and $\mathbf{r_1} \neq \mathbf{r_2}$. In this case, the larger requirement set $\mathbf{r_2}$ could be ignored in principle, and an initiative queue can be allocated for $\mathbf{r_1}$ only. Then, an initiative token j for an evicted logical actor instance a_j which would normally be enqueued to the initiative queue allocated for $\mathbf{r_2}$ is instead enqueued to the initiative queue allocated for $\mathbf{r_1}$. Note that it is also possible to determine requirement sets that do not correspond to transitions of the logical actor instances. In particular, a requirement set $\mathbf{r}_\perp = \mathbf{0}$ may be used to replace any other requirement set $\mathbf{r} > \mathbf{r}_\perp$. In this case, only a single initiative queue would be required. However, all information about resource requirements is lost in this case. The allocation of initiative queues therefore can be seen as an optimization problem which may be studied in future work.

Example 6.7. The partial order "$<$" over the requirement sets from Table 6.3a is shown in Table 6.3b. In particular, $\mathbf{r_2} < \mathbf{r_3}$. For the example, an initiative queue is allocated for requirement sets $\mathbf{r_1}$, $\mathbf{r_2}$, $\mathbf{r_4}$, and $\mathbf{r_5}$.

When a logical actor instance a_j is evicted from a reservation station, an initiative token j is enqueued to the initiative queue(s) according to the current actor mode. Thus, in principle, a logical actor instance may be contained in more than one initiative queue. In order to ensure that a_j is only enqueued once in a given initiative queue, a bit is stored alongside the actor state which indicates whether a_j is enqueued in the initiative queue. This bit is set when the initiative token j is enqueued to the initiative queue, and cleared when the initiative token j is dequeued from the initiative queue (and the actor state has been fetched from the backing store).

Example 6.8. For the example, four initiative queues have been allocated. Thus, an evicted EP may be enqueued in up to four initiative queues. In this case, four bits are allocated alongside the actor state of each logical actor instance in order to remember this information. For 2^{24} EPs, this requires an additional 8 MB of memory in the backing store. In contrast, as implemented, the actor state amounts to 107 bytes for one EP, i.e., a total of approx. 1.7 GB of memory is required in the backing store.

An initiative token j which becomes visible on an initiative queue is immediately dequeued if the corresponding logical actor instance a_j is already bound to a reservation station, analogously to colored tokens on dispatch channels. Otherwise, an *idle* reservation station must be determined. In this case, however, the semaphore channels and the outgoing channels of an idle RS are checked whether enough tokens and free places are available according to the requirements associated with the initiative queue in question. Note that the initiative token is only dequeued from the initiative queue if this is the case. Otherwise, the initiative token is not dequeued.

Overall, colored tokens on dispatch channels and initiative tokens on initiative queues represent token sources from which logical actor instances are determined that have to be bound to reservation stations. As there are typically multiple dispatch channels and initiative queues, the question arises which token source should be considered next if there are any idle reservation stations. Here, a possible scheduling scheme consists in allocating some *credits* for each token source. The number of reservation stations then corresponds to the total number of allocated credits. A token of color j is only dequeued from a token source (and possibly forwarded to an RS) if either the logical actor instance a_j is already bound to an RS and has already acquired a credit for the token source, or if the token source has some credits left. In case of the latter, a credit is consumed for the token source and associated with the RS to which the token is forwarded. This credit is returned when the RS becomes idle. While at least one credit must be allocated for each token source, allocating more than one credit allows more logical actor instances to be processed concurrently. Other scheduling schemes may be studied in future work.

Example 6.9. For the InfiniBand example, at least six reservation stations must be allocated if the credit-based scheduling scheme is used: One RS for each dispatch channel, and four reservation stations for the four initiative queues.

Mapping of Reservation Stations to Physical Actor Instances

Compared to the input model where the logical actor instances would be synthesized as dedicated entities, the mapping of logical actor instances to a few reservation stations already results in significant resource savings (or may even enable synthesis in the first place). However, it can be observed that not all guard functions and action functions of the reservation stations are active at the same time in the general case. Thus, it is typically possible to reduce the number of instantiated guard functions and action functions without significantly degrading the overall performance of the synthesized model (cf. [ZHF+14; ZHFT12a]). In the following, a synthesized guard function or action function is referred

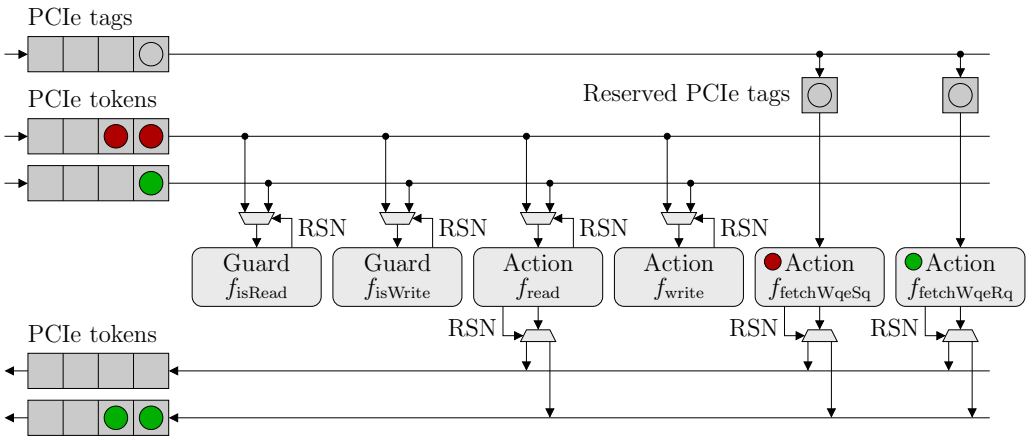


Figure 6.11: Token forwarding to/from functional units. For the sake of clarity, channels for IB tokens and link credits have been omitted.

to as functional unit (FU). A physical actor instance therefore can be seen as instantiating one FU for each guard function and action function of the actor in question. While more than one physical actor instance could be instantiated in principle, a more fine-grained allocation of FUs is also possible in order to eliminate bottlenecks. In order to provide for a fine-grained utilization of FUs, each FU can be individually configured with the desired reservation station number (RSN) for which the FU should be executed.

The dynamic mapping of reservation stations to functional units is performed according to the operational semantics outlined in Sections 3.1.1 and 5.1.1. During the evaluation phase of a transition t, the resources (i.e., tokens and free places) required by t are checked against the available resources.

For colored tokens from dispatch channels, the number of available tokens is equal to the number of tokens available in the corresponding FIFO channel to the RS. (Remember that these FIFO channels contain only tokens of the same color.) Analogously, for tokens produced on merge channels, the number of available free places is equal to the number of free places available in the corresponding FIFO channels from the RS.

Example 6.10. Consider Figure 6.10 and Figure 6.11: Two PCIe tokens and no IB tokens are currently available for EP_{red}. In contrast, one PCIe token and one IB token are currently available for EP_{green}.

In contrast to colored tokens, colorless tokens cannot be assigned to a logical actor instance when they appear at the composite actor. In fact, doing so may introduce deadlocks into the system. Instead, colorless tokens can only be assigned to a logical actor instance when a transition is about to be executed. To this end,

if a transition t is to be executed, the requested tokens according to $\text{cons}(p, t.f_a)$ can be copied to a dedicated FIFO buffer allocated for the FU corresponding to the action function $t.f_a$. While the reserved tokens are copied, the semaphore channel appears empty to other guard functions and action functions in order to avoid token conflicts. Note that guard functions which have been evaluated earlier may have peeked at the tokens which are now copied to the FU. The corresponding conflict-resolution scheme is discussed later. For tokens produced on semaphore channels, it is assumed that enough free places are always available.

Example 6.11. Considering Figure 6.11, $f_{\text{fetchWqeSq}}$ is currently executed for RS_1, while $f_{\text{fetchWqeRq}}$ is executed for RS_2. While one tag token has been reserved and copied to each action FU, both action FUs have not yet consumed their reserved token. For other guard functions and action functions, one tag token is still available in the semaphore channel.

Assume that for a transition t of a reservation station RS_i enough tokens are available according to the values of peek. Then, the guard function $t.f_g$ can be evaluated. The corresponding FU can be executed for RS_i if it is not currently executed for any RS. In this case, the FU is configured with the RSN i, and execution is started. After the execution of the FU is finished, the result is *cached* in RS_i. This prevents multiple evaluations of $t.f_g$ for RS_i, and allows another transition t' with $t'.f_g = t.f_g$ to reuse the cached result of $t.f_g$.

The action function $t.f_a$ can be executed if the cached result of $t.f_g$ is $\top$, and enough tokens and free places are available according to the values of cons and prod. The corresponding FU can be executed for RS_i if it is not currently executed for any RS, and no action function is currently executed for RS_i. In this case, the FU is configured with the RSN i, and execution is started. Executing $t.f_a$ has the following consequences:

- All guard function results cached in RS_i are invalidated, as they must be re-evaluated after the action function has been executed.
- All FUs which are currently executed for RS_i are reset, thereby freeing the FUs for different reservation stations. Note that these reset FUs correspond to guard functions.
- RS_i is blocked from starting any other FU until the action function has been executed.
- For other reservation stations than RS_i, cached guard function results are invalidated and active FUs corresponding to guard functions are reset, but only if the guard function may have peeked at tokens that are now reserved for the executed action function.

Let $I_{\text{sem}} \subseteq I$ be the subset of input ports of a logical actor instance connected to semaphore channels. Remember that only colorless tokens can induce conflicts between reservation stations. Then, the subset of guard functions $\hat{F}_g(f_a) \subseteq F_g$ that are invalidated by an action function $f_a \in F_a$ can be statically determined as follows: $\hat{F}_g(f_a) = \{f_g \in F_g \mid \exists p \in I_{\text{sem}} : \text{cons}(p, f_a) > 0 \wedge \text{peek}(p, f_g) > 0\}$, i.e., guard functions which peek at some tokens on an input port p from which the action function f_a consumes some tokens.

Example 6.12. In the InfiniBand example, the set of invalidated guard functions is empty for all action functions.

The dynamic mapping of reservation stations to physical actor instances must satisfy Requirement 3.1 on page 32. For colored tokens, this requirement is trivially satisfied by the fact that only a single action function may be active for the corresponding RS, and the invalidation of cached guard function results when an action function is executed for the RS. For colorless tokens, assume two transitions t_1 and t_2 which are evaluated and executed for different reservation stations, and that $t_1.f_g \in \hat{F}_g(t_2.f_a)$ and $t_2.f_g \in \hat{F}_g(t_1.f_a)$. Without loss of generality, assume that $t_1.f_a$ is executed before $t_2.f_a$. In this case, the peeked tokens are copied into the FIFO channel to the FU which corresponds to $t_1.f_a$ before $t_1.f_a$ is executed. Additionally, the cached results of $t_1.f_g$ and $t_2.f_g$ are cleared. Then, $t_2.f_a$ cannot be executed until $t_2.f_g$ has been re-evaluated, which can only happen after the reserved tokens (including the peeked tokens) have been removed from the corresponding semaphore channel. Thus, Requirement 3.1 is satisfied.

The actor variables of a logical actor instance are stored alongside the RS where the logical actor instance is bound to, and can be accessed by FUs accordingly. Concerning hazards w.r.t. the actor variables, write-after-write (WAW) hazards are prevented as only one action function may be executed for a given RS at any one time. Note that only action functions may modify the actor variables. Read-after-write (RAW) hazards are avoided because no FUs are started for a given RS while an action function is executed for the RS. Write-after-read (WAR) hazards are resolved by resetting all FUs which are currently executed for a given RS when an action function is to be executed for the RS.

Results

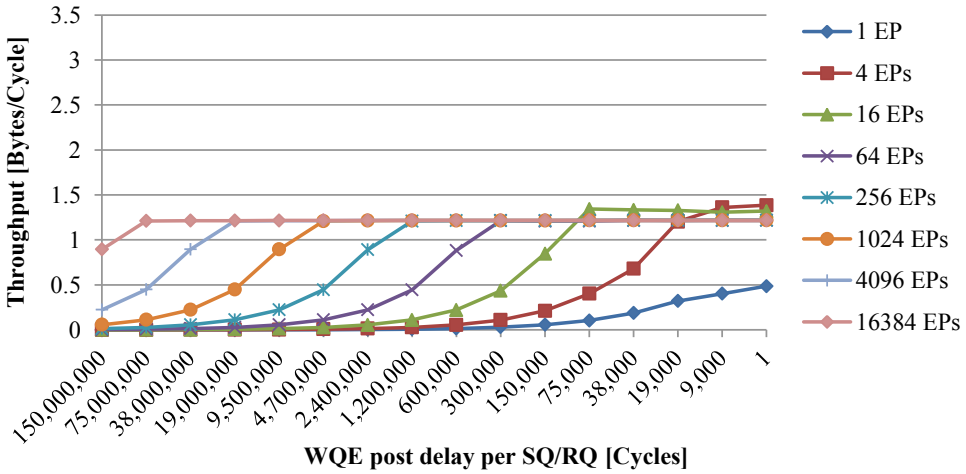
In order to show that the proposed inter-process resource sharing approach can multiplex a large number of logical actor instances without introducing deadlocks into the system, it has been applied to the InfiniBand network controller. To this end, a single EP and the token coloring actors have been implemented within the proposed dataflow model. Subsequently, the multiplexing logic has been semi-automatically generated for the single EP as described in the previous sections.

Two variants have been generated: 1) a functional implementation realized as a hierarchical actor within the proposed dataflow model, and 2) a bus-cycle-accurate SystemC RTL model. While the former is used for functional verification of the proposed multiplexing approach, the latter is used to obtain performance estimations, and could be used for subsequent HLS steps as described in Section 6.3.2. In the following, only the SystemC model is discussed in more detail.

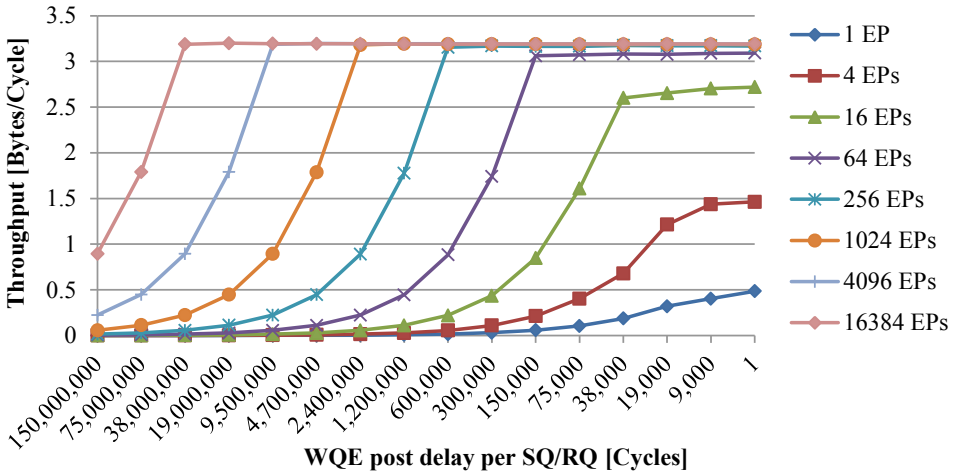
In order to model the environment, certain assumptions about the target platform have been made. Considering a Xilinx Virtex 6 FPGA platform, the various latencies are given in Table 6.4. The EP state consists of the actor variables (105 bytes), the current actor mode (16 bits), and the bits which indicate the initiative queues where an EP is enqueued to (4 bits). Thus, for 2^{24} EPs, a backing store of ca. 1.7 GB is required for the overall EP state, which is therefore assumed to be bound to an off-chip memory. Note that this off-chip memory has a data path of 512 bits in case of the target platform. Thus, the state of an EP can be written in two cycles. In order to hide the off-chip memory read latency of 60 cycles, the implementation uses a direct-mapped cache for the EP state. Here, 24 BRAM blocks (with 36 kbit memory each) can be used to cache the state of 1024 EPs. In order to provide for a fast access of actor variables, these are assumed to be kept in registers in each RS. Finally, for each

Parameter	**Value**
Active EPs	1, 4, 16, 64, 256, 1024, 4096, 16384
WQEs per SQ/RQ	64
WQE size	1024 bytes (256 tokens)
IB/PCIe MTU size	256 bytes (64 tokens)
WQE post delay per SQ/RQ	150,000,000 cycles – 1 cycle
Credits per dispatch channel	1, 4
Credits per initiative queue	1
Reservation stations	6, 12
FUs per guard/action function	1
Size of EP state cache	1024 EPs
BRAM/FIFO write latency	1 cycle
BRAM/FIFO read latency	1 cycle
Off-chip mem. write latency (EP state)	1 cycle
Off-chip mem. read latency (EP state)	60 cycles
Main mem. write latency (PCIe)	1 cycle
Main mem. read latency (PCIe)	60 cycles
Link delay	20000 cycles

Table 6.4: Simulation/Architecture parameters



a) Throughput for 6 reservation stations.



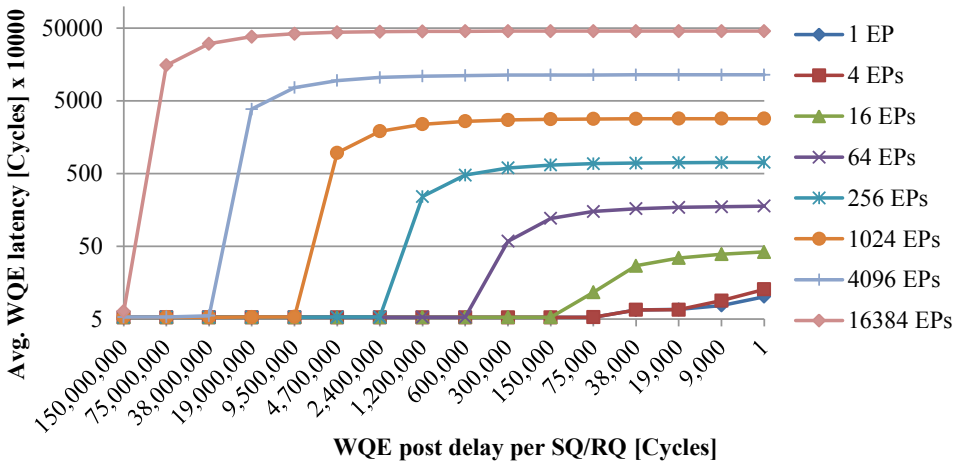
b) Throughput for 12 reservation stations.

Figure 6.12: Results for the simulation parameters given in Table 6.4.

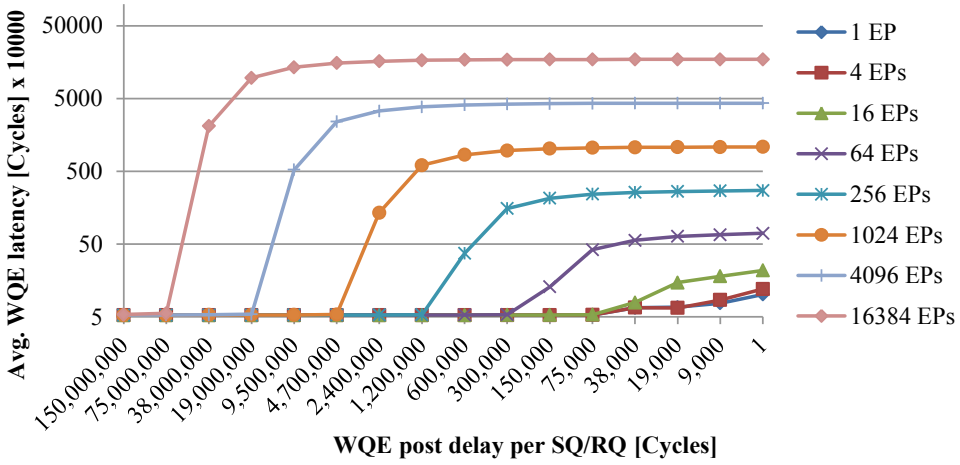
FU, a SystemC process has been instantiated. This allows subsequent HLS tools to generate concurrent modules for each FU (cf. Section 6.3.3).

The various parameters of the evaluated test cases are also given in Table 6.4. For each test case, a certain number of EPs is initialized prior to the posting of WQEs. Then, WQEs are simultaneously posted to the SQ of an EP and to the RQ of its peer EP after the WQE post delay has elapsed for the SQ. A WQE is *retired* when the corresponding CQE has been generated by the associated EP. The simulation is finished when all WQEs have been retired.

The simulation results are shown in Figure 6.12 (throughput) and in Figure 6.13 (latency). On the one hand, *throughput* denotes the ratio between the total



a) Latency for 6 reservation stations.



b) Latency for 12 reservation stations.

Figure 6.13: Results for the simulation parameters given in Table 6.4.

number of bytes transferred between the EPs and the total number of cycles of the simulation. On the other hand, *avg. WQE latency* denotes the average number of cycles from the posting of a WQE to the receiving of the corresponding CQE. The following observations can be made:

- The WQE latency does not approach infinity because only a limited number of 64 WQEs is posted to each SQ/RQ.

- A single EP cannot fully utilize the functional units even if the WQE post frequency is increased, resulting in a low throughput.

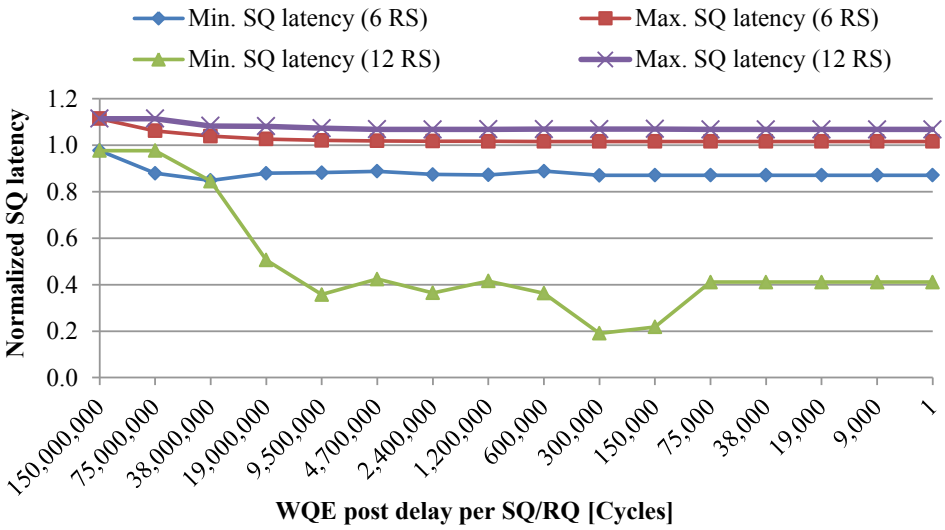


Figure 6.14: Normalized SQ latencies for 16384 EPs. The RQ latencies are virtually identical and are therefore not shown.

- For the minimum number of 6 reservation stations, the throughput for 4 and 16 EPs is slightly better than the throughput for more EPs. As only one reservation station is allocated per dispatch channel in this case, EPs have to be evicted more often from the reservation stations for more than 16 EPs. Eliminating this bottleneck by allocating four reservation stations per dispatch channel improves both peak throughput and latency by a factor of ca. 2.6 for test cases with more than 16 EPs, because more EPs can now be executed concurrently by the FUs.

- The peak throughput can be sustained even for a large number of active EPs in case of 12 reservation stations. This shows that only a small overhead is incurred by the dynamic scheduling scheme based on dispatch channels and initiative queues.

- While easily accomplished, increasing the number of FUs has only a negligible effect on the peak throughput of the generated bus-cycle-accurate SystemC model with a 32 bit data path, as the peak throughput of ca. 3.2 bytes/cycle for the minimum number of FUs already approaches the theoretical maximum throughput of 4 bytes/cycle. Note that after HLS, the FUs may require more cycles compared to the bus-cycle-accurate model. In this case, increasing the number of FUs may be required in order to eliminate some bottlenecks.

Figure 6.14 shows the normalized minimal and maximal SQ latencies for 16384 active EPs. Note that the RQ latencies are virtually identical to the SQ latencies

and are therefore not shown. Here, SQ/RQ latency refers to the average number of cycles from the posting of the first WQE to the SQ/RQ to the receiving of the CQE corresponding to the last WQE posted to the SQ/RQ. Note that for each WQE post delay, the resulting minimal and maximal SQ/RQ latencies have been normalized w.r.t. their respective average latency obtained for the WQE post delay, which is therefore not shown. It can be observed that the maximal SQ/RQ latency is at most 8% larger than the average latency in case of 6 reservation stations, and at most 11% larger in case of 12 reservation stations. Furthermore, it can be observed that the minimum SQ/RQ latency is at most 15% smaller than the average latency in case of 6 reservation stations, and at most 80% smaller than the average latency in case of 12 reservation stations.

Here, the initiative queues which are basically FIFO channels apparently provide for a fair arbitration of logical actor instances which are blocked on some missing resources. In case of 12 reservation stations, the small minimum latency may stem from the fact that four reservation stations are allocated for each dispatch channel, and that the implemented selection strategy of idle reservation stations always selects the first one. Thus, some EPs may reside longer in reservation stations without being evicted, and can therefore process their WQEs faster than other EPs. Here, a different selection scheme of idle reservation stations like LRU may improve the average latencies.

Finally, the Xilinx Vivado HLS tool has been evaluated in order to further synthesize the generated bus-cycle-accurate SystemC model. To this end, the FUs have been successfully synthesized to RTL in isolation. Note that results for the synthesis of a bus-cycle-accurate model with similar FUs have been presented in Section 6.3.3 in the context of the parallel evaluation of transitions. Next, the remaining data path (cf. Figure 6.10 and Figure 6.11) and the overall control logic which performs the mapping of logical actor instances to reservation stations and the mapping of reservation stations to FUs should have been synthesized by the HLS tool. However, due to the control-flow-dominated nature of the control logic, the Vivado HLS tool was not able to synthesize the control logic. As a time-consuming manual translation of the control logic to RTL is beyond the scope of this thesis, the synthesis of the bus-cycle-accurate SystemC model therefore has not been pursued any further.

In the general case, the inter-process resource sharing approach extends the design space by providing for an automatic mapping of n logical actor instances to m physical actor actor instances. Thus, it is possible to choose a desired number of $1 \leq m \leq n$ physical actor actor instances, e.g., based on the evaluation of the generated bus-cycle-accurate model. In particular, the simulation results of the bus-cycle-accurate InfiniBand model show that a large number of logical actor instances can be bound to a single physical actor instance. Instantiating more physical actor instances in order to eliminate bottlenecks is then easily achieved during design space exploration at the algorithmic/instruction level.

6.4 Related Work and Limitations

Daedalus [TNS+07; NSD08] is a system-level design flow which starts by translating sequential C programs into a platform-independent Kahn process network (KPN) (cf. Section 4.8) suitable for design space exploration. However, the input C programs are restricted to static affine nested loop programs (SANLPs). While our proposed design flow requires the user to provide a concurrent (dataflow) model as input, the model is not restricted to KPNs, and the functionality of the application is not restricted to loop programs. The resulting dataflow model is then synthesized by the ESPAM tool [SDN06] which is similar to the proposed synthesis framework. However, in contrast to our proposed design flow, the synthesis at system level is not separated from the synthesis at the algorithmic/task level.

Koski [KKO+06] is a multiprocessor System-on-Chip (MPSoC) design flow based on UML 2.0. Comparable to the hierarchical model described in Section 5.1, components can be hierarchically nested, and are divided into structural components and functional components. The behavior of functional components is modeled by Statecharts [Har87], comparable to the proposed dataflow model (cf. Section 3.2). Components communicate with other components by means of ports which are connected by signals. While not formally described, components adhere to the KPN model of computation (MoC). Actors in the proposed dataflow model are not restricted to KPN semantics.

The System-on-Chip Environment (SCE) introduced in [DGP+08] is based on SpecC [GZD+00] (cf. Section 2.1). In contrast to the proposed approach, actors implemented by means of SpecC are not restricted to communication via channels with FIFO semantics. While SCE therefore supports the automatic refinement of the model during system synthesis, (automatic) decision-making is difficult [GHP+09]. In order to enable model-based analysis and optimizations in principle, actors implemented by means of SpecC may be transformed into the proposed dataflow model as described in Section 2.3.

The Oldenburg System Synthesis Subset (OSSS) [GON+08] is based on a synthesizable subset of SystemC. While actors can be synthesized to software or hardware, no explicit support for model-based optimizations is included. Again, in order to enable model-based analysis and optimizations in principle, it may be possible to transform actors implemented by means of the OSSS into the proposed dataflow model as described in Section 2.3. In this context, [GKO+10] describes how methods of user-defined SystemC primitive channels can be transformed into guarded actions.

PeaCE (Ptolemy extension as a Codesign Environment) [HKL+08] provides a design flow for multimedia applications with real-time constraints. PeaCE classifies actors into *signal processing actors* and *control actors*. While signal processing actors are limited to the SDF MoC, control actors are described

by FSMs. In contrast, the proposed dataflow model unifies signal processing actors and control actors by providing for dynamic dataflow (DDF) actors which are described by FSMs. Besides asynchronous communication, PeaCE supports synchronous communication between actors, which basically corresponds to FIFO channels of size 0 (which are also known as *rendezvous channels*). For the synthesis framework presented in Section 6.1, the support of rendezvous channels could be considered as part of future work.

While more and more industrial design flows rely on HLS tools as can be seen by the broad availability of commercial HLS tools [MVG+13; For14; Cad14; Cal14; NEC14; Xil14], they do not support model-based optimizations. In particular, global optimizations like resource sharing across process boundaries are typically not supported.

HLS approaches like [ALP95; WL96; HM99; JH04] concentrate on static dataflow MoCs. In this context, some inter-process resource sharing approaches are reported for identical SDF actors, comparable to the binding of reservation stations to physical actor instances described in Section 6.3.4. However, the proposed inter-process resource sharing approach not only targets more general DDF graphs, but additionally considers the binding of a large number of logical actor instances to a smaller number of reservation stations.

In [JL98], it is shown how resources can be shared within a process group such that less than one resource per operation type and process is required. Basically, a periodic authorization function is defined for each global resource which permits processes to access a shared resource during fixed time slots. Note that while this approach permits static scheduling using a modified list scheduling scheme, time slots cannot be reused if the corresponding process is not active. The inter-process resource sharing approach presented in Section 6.3.4 uses a dynamic scheduling scheme which is able to efficiently capture the fact that actors may not always be active. However, this is an orthogonal approach, and the fine-grained resource sharing scheme from [JL98] may well be used to perform resource sharing between the functional units used in the proposed approach (cf. Figure 6.5 on page 142).

More recently, [SWN07] introduces resource sharing in the context of pipeline scheduling. Here, the goal is to construct pipelined schedules that are expected to meet a particular target throughput. However, the approach only addresses SDF graphs. The inter-process resource sharing approach presented in Section 6.3.4 targets more general DDF graphs.

In [JMP+11], the HLS of CAL actors (cf. Section 3.4) is described. Similar to the bus-cycle-accurate model (cf. Section 6.3.2) generated by the proposed synthesis framework where transitions can be evaluated in parallel, actions of CAL actors are synthesized to distinct processes which provides for a parallel evaluation of the input patterns and guard expressions. However, no inter-process optimizations are reported.

7 Conclusions and Future Work

Heterogeneous multiprocessor System-on-Chip (MPSoC) architectures are becoming more and more important in embedded systems in order to satisfy the ever-increasing computational demands of functions to be performed by these systems. Thus, at system level, choosing the right programming model is a challenging task. In this thesis, dataflow models of computation (MoCs) have been considered, which are typically used to model streaming applications as commonly found, e.g., in the multimedia or networking domain. In a dataflow model, concurrent actors communicate and synchronize via tokens transmitted over channels.

As a key result, the proposed dataflow model provides for a *seamless model-based design flow* from the system level to the instruction/logic level for a wide range of streaming applications. At system level, *binding and scheduling decisions are incorporated into the model by means of hierarchical actors* which have the same dataflow semantics as leaf actors. In turn, this enables the same *analysis and optimization techniques* like the identification of less expressive dataflow MoCs to be applied to leaf actors and hierarchical actors. Incorporating scheduling decisions into the model allows *a wide range of scheduling strategies to be synthesized* at the task/algorithmic level in principle. As the proposed hierarchical dataflow model is used as input to subsequent synthesis steps at the task/algorithmic level, *complex model-based optimizations* like the inter-process resource sharing approach can still be applied at these lower levels of abstraction.

Expressiveness vs. Analyzability

In order to support a wide range of streaming applications, actors are described by means of a finite state machine (FSM) whose transitions are based on guarded actions and exhibit a static communication behavior. Thus, in contrast to existing dataflow models, the proposed dataflow model provides for dynamic (i.e., nondeterministic) actors, while the static communication behavior of transitions still enables the identification of less expressive dataflow MoCs. In turn, this enables the application of domain-specific analysis and optimization techniques

which have been developed for these restricted dataflow MoCs, like the static scheduling of SDF and CSDF actors.

Concurrency Model vs. Actor Semantics

It has been shown that the proposed dataflow model can be automatically derived from existing (well-formed) SystemC models, and that a wide range of scheduling strategies can be subsequently incorporated into the model at system level. The expressiveness of the proposed dataflow model has been chosen as restricted as possible in order to adequately achieve these objectives. In particular, the proposed dataflow model cleanly separates the concurrency model from the FSM semantics: Actors are executed concurrently, whereas the transitions of a given actor are executed in a sequential manner. While more expressive dataflow MoCs exist which provide for the non-sequential execution of transitions, the sequential semantics of transitions provide for an efficient hardware/software synthesis of actors. Moreover, the sequential semantics of transitions induce the same well-defined semantics to AND modes of hierarchical FSMs, which can be used to cope with the complexity of real-world applications as outlined for an InfiniBand network adapter in Section 6.3.

Model-Based Optimizations

Applying model-based micro-architectural optimizations like token caching and the parallel evaluation of guards, the throughput of a JPEG decoder synthesized to hardware could be improved from 40 frames per second (FPS) to 100 FPS. In the context of inter-process resource sharing, an InfiniBand network adapter consisting of 2^{24} logical actor instances could be successfully multiplexed onto a single physical actor instance. As these optimizations are automatically performed based on the proposed dataflow model, the overall modeling complexity is greatly reduced. Moreover, these model-based optimizations considerably extend the design space, as different configurations can be automatically synthesized and evaluated during design space exploration in principle.

7.1 Future Work

Dynamic Model

It can be observed that the proposed dataflow model is static, i.e., model components like actors and channels cannot be instantiated at run time. However, as action functions are considered to be black boxes, the proposed dataflow model provides for the dynamic instantiation of actors in principle. In the context of hardware synthesis, the dynamic instantiation of components is difficult to

achieve (it may be possible using partial reconfiguration features offered by FPGAs). However, the proposed inter-process resource sharing approach could be used to dynamically instantiate actors even in hardware, as only the physical actor instances have to be synthesized, and the logical actor instances are multiplexed onto these physical actor instances at run time. In the context of software synthesis, a dynamic instantiation of components could provide for a more efficient utilization of computing resources in future many-core embedded systems [THH+11].

Scheduling Actors Bound to Hardware

Typically, only actors bound to the same processor are scheduled, as they cannot be executed concurrently. However, future work may evaluate the scheduling of actors bound to hardware. In this case, however, the goal is not to reduce the scheduling overhead, which can be considered to be nonexistent for actors bound to hardware, as they are evaluated and executed concurrently in the absence of any scheduling strategy. Instead, it can be expected that resource utilization is reduced, as resources can then be shared between actors. The feasibility of scheduling actors bound to hardware has been shown by the proposed inter-process resource sharing approach.

Model-Based Representation of Transaction Schedules

The scheduling of transactions bound to the same communication resource has only been casually addressed in the context of the periodic partial order scheduling approach. While a more comprehensive model-based representation of transaction schedules may be studied by future work, the emergence of networks on chip (NoCs) as the prevalent communication resource in MPSoCs suggests that transactions initiated by many applications are bound to the same NoC. However, as these applications are typically executed independently of each other, a model-based representation of transaction schedules seems difficult in this case. It can therefore be assumed that the scheduling of transactions bound to the same NoC is preferably performed by the communication resource in question, typically in a dynamic manner, analogously, e.g., to concurrent memory accesses which are typically dynamically scheduled by memory modules.

Non-Functional Scheduling Strategies

The hierarchical dataflow model provides for the efficient representation of a wide range of scheduling strategies. However, only functional scheduling schemes have been considered. The integration of non-functional scheduling schemes like time-driven scheduling or preemptive scheduling could be studied by future work.

Mixed Hardware/Software Synthesis

Finally, the proposed synthesis framework provides for a mixed hardware/software synthesis in principle. However, as this thesis focuses on model-based optimizations, the hardware synthesis and software synthesis of the proposed dataflow model have only been evaluated in isolation. The evaluation of a mixed hardware/software synthesis could be conducted by future work.

Bibliography

[Acc04] Accellera Systems Initiative, *SystemVerilog 3.1a Language Reference Manual: Accellera's Extensions to Verilog®*, 2004.

[Acc12] Accellera Systems Initiative, *IEEE Standard 1666-2011 SystemC Language Reference Manual*. IEEE Standards Association, 2012.

[ALP95] M. Ade, R. Lauwereins, and J. A. Peperstraete, "Hardware-Software Codesign with GRAPE", in *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, 1995, pp. 40–47.

[ANRD04] Arvind, R. S. Nikhil, D. L. Rosenband, and N. Dave, "High-level Synthesis: An Essential Ingredient for Designing Complex ASICs", in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2004, pp. 775–782.

[AS11] S. H. Attarzadeh Niaki and I. Sander, "Semi-Formal Refinement of Heterogeneous Embedded Systems by Foreign Model Integration", in *Proceedings of the Forum on Design Languages (FDL)*, 2011, pp. 1–8.

[BAS12] G. S. Beserra, S. H. Attarzadeh Niaki, and I. Sander, "Integrating Virtual Platforms into a Heterogeneous MoC-Based Modeling Framework", in *Proceedings of the Forum on Design Languages (FDL)*, 2012, pp. 143–150.

[BB00a] B. Bhattacharya and S. Bhattacharyya, "Quasi-static Scheduling of Reconfigurable Dataflow Graphs for DSP Systems", in *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, Jun. 2000, pp. 84–89.

[BB00b] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized Dataflow Modeling of DSP Systems", in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2000, pp. 3362–3365.

[BBDV06] A. Bertolino, A. Bonivento, G. De Angelis, and A. S. Vincentelli, "Modeling and Early Performance Estimation for Network Processor Applications", in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2006, pp. 753–767.

[BBHL95] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, "Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms", *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 42, no. 3, pp. 138–150, Mar. 1995.

[BBS11] D. Baudisch, J. Brandt, and K. Schneider, "Translating Synchronous Systems to Data-Flow Process Networks", in *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2011, pp. 354–361.

[BC12] O. Bouissou and A. Chapoutot, "An Operational Semantics for Simulink's Simulation Engine", in *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, ACM, 2012, pp. 129–138.

[BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-Static Dataflow", *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, Feb. 1996.

[BFM+05] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, and H.-J. Wolff, "Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, IEEE Computer Society, Mar. 2005, pp. 1044–1049.

[BG92] G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation", *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[BGJ+97] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and K. Suzuki, *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[BHLP09] Y. Bertot, G. Huet, J.-J. Lvy, and G. Plotkin, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, first edition. Cambridge University Press, 2009.

[BKKB02] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya, "Intermediate Representations for Design Automation of Multiprocessor DSP Systems", in *Design Automation for Embedded Systems*, Kluwer Academic Publishers, 2002, pp. 307–323.

[BL93] S. S. Bhattacharyya and E. A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping", *Journal of VLSI Signal Processing Systems*, vol. 6, no. 3, pp. 271–288, 1993.

[BLM00] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software Synthesis and Code Generation for Signal Processing Systems", *IEEE Transactions on Circuits and Systems*, vol. 47, no. 9, Sep. 2000.

[BLM96] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[BML97] S. S. Bhattacharyya, P. Murthy, and E. Lee, "APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations", *Design Automation for Embedded Systems*, Jan. 1997.

[BRM+12] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen, "Pause 'n' Play: Formalizing Asynchronous C#", in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer, 2012, pp. 233–257.

[BSS10] J. Brandt, K. Schneider, and S. K. Shukla, "Translating Concurrent Action Oriented Specifications to Synchronous Guarded Actions", in *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, ACM, 2010, pp. 47–56.

[Buc93] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model", PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Sep. 1993.

[Buc94] J. T. Buck, "Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer-Valued Control Streams", in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 1994, pp. 508–513.

[Cad14] Cadence, *C-to-Silicon Compiler*, `http://www.cadence.com/products/sd/silicon_compiler`, 2014.

[Cal14] Calypto, *Catapult*, `http://calypto.com/en/products/catapult/overview`, 2014.

[CHEP71] F. Commoner, A. W. Holt, S. Even, and A. Pnueli, "Marked Directed Graphs", *Journal of Computer and System Sciences*, vol. 5, no. 5, pp. 511–523, 1971.

[CPB12] I. Chukhman, W. Plishker, and S. Bhattacharyya, "Instrumentation-Driven Model Detection for Dataflow Graphs", in *Proceedings of the International Symposium on System-on-Chip (SoC)*, 2012, pp. 1–8.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A Declarative Language for Real-time Programming", in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 1987, pp. 178–188.

[DGP+08] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski, "System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design", *EURASIP Journal on Embedded Systems (JES)*, vol. 2008, 5:1–5:13, Jan. 2008.

[dKES+00] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers, "YAPI: Application Modeling for Signal Processing Systems", in *Proceedings of the Design Automation Conference (DAC)*, Jun. 2000, pp. 402–405.

[DSB+12] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Modeling Static-Order Schedules in Synchronous Dataflow Graphs", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2012, pp. 775–780.

[DZ83] J. D. Day and H. Zimmermann, "The OSI Reference Model", *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334–1340, 1983.

[Ecm06] Ecma International, *Standard ECMA-334*, `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, 2006.

[EJ03] J. Eker and J. Janneck, "CAL Language Report: Specification of the CAL Actor Language", Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California, Tech. Rep., 2003.

[EJL+03] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming Heterogeneity - The Ptolemy Approach", *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[ET05] S. A. Edwards and O. Tardieu, "SHIM: A Deterministic Model for Heterogeneous Embedded Systems", in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, 2005, pp. 264–272.

[FKH+08] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "A Generalized Static Data Flow Clustering Algorithm for Mpsoc Scheduling of Multimedia Applications", in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, ACM, Oct. 2008, pp. 189–198.

[For14] Forte Design Systems, *Cynthesizer*, `http://www.forteds.com/products/cynthesizer.asp`, 2014.

[GCKR12] A. Gerstlauer, S. Chakravarty, M. Kathuria, and P. Razaghi, "Abstract System-Level Models for Early Performance and Power Exploration", in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[GGL12] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation", *Parallel Processing Letters*, 2012.

[GHP+09] A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, Oct. 2009.

[GK83] D. Gajski and R. H. Kuhn, "New VLSI Tools - Guest Editors Introduction", *IEEE Computer*, vol. 16, no. 12, pp. 11–14, 1983.

[GKO+10] K. Grüttner, H. Kleen, F. Oppenheimer, A. Rettberg, and W. Nebel, "Towards a synthesis semantics for systemc channels", in *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, ACM, 2010, pp. 163–172.

[GLL99] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 742–760, 1999.

[GLMS02] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GON+08] K. Grüttner, F. Oppenheimer, W. Nebel, F. Colas-Bigey, and A.-M. Fouilliart, "SystemC-based Modelling, Seamless Refinement, and Synthesis of a JPEG 2000 Decoder", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2008, pp. 128–133.

[GS13a] M. Gesell and K. Schneider, "Modular Verification of Synchronous Programs", in *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD)*, IEEE Computer Society, 2013, pp. 70–79.

[GS13b] M. Gesell and K. Schneider, "Translating Synchronous Guarded Actions to Interleaved Guarded Actions", in *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2013, pp. 167–176.

[GZD+00] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[Har87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[HB07] C. Hsu and S. S. Bhattacharyya, "Cycle-Breaking Techniques for Scheduling Synchronous Dataflow Graphs", Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep., Feb. 2007.

[HHB+12] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hubner, R. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe, "Invasive Manycore Architectures", in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012, pp. 193–200.

[HHP+07] K. Huang, S. I. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.-I. Chae, L. Carro, and A. A. Jerraya, "Simulink-Based MPSoC Design Flow: Case Study of Motion-JPEG and H.264", in *Proceedings of the Design Automation Conference (DAC)*, 2007, pp. 39–42.

[HKB05] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software Synthesis from the Dataflow Interchange Format", in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Sep. 2005, pp. 37–49.

[HKL+08] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "PeaCE: A Hardware-software Codesign Environment for Multimedia Embedded Systems", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, 24:1–24:25, May 2008.

[HM99] J. Horstmannshoff and H. Meyr, "Optimized System Synthesis of Complex RT Level Building Blocks from Multirate Dataflow Graphs", in *Proceedings of the International Symposium on System Synthesis (ISSS)*, 1999, pp. 38–43.

[HMT06] A. Habibi, H. Moinudeen, and S. Tahar, "Generating Finite State Machines from SystemC", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, IEEE Computer Society, Mar. 2006, pp. 76–81.

[HN12] M. Hosseinabady and J. Nunez-Yanez, "Run-time Stochastic Task Mapping on a Large Scale Network-on-Chip with Dynamically Reconfigurable Tiles", *IET Computers & Digital Techniques*, vol. 6, no. 1, pp. 1–11, 2012.

[Hoa85] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall, Inc., Apr. 1985.

[HSV04] F. Herrera, P. Sánchez, and E. Villar, "Languages for system specification", in, C. Grimm, Ed., Kluwer Academic Publishers, 2004, ch. Modeling of CSP, KPN and SR Systems with SystemC, pp. 133–148.

[HT12] A. S. Hartman and D. E. Thomas, "Lifetime Improvement Through Runtime Wear-based Task Mapping", in *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, ACM, 2012, pp. 13–22.

[HTS+06] A. Habibi, S. Tahar, A. Samarah, D. Li, and O. A. Mohamed, "Efficient Assertion Based Verification using TLM", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, IEEE Computer Society, Mar. 2006, pp. 106–111.

[Inf14] InfiniBand Trade Association, *InfiniBand Specification*, `http://www.infinibandta.org`, 2014.

[JH04] H. Jung and S. Ha, "Hardware Synthesis From Coarse-Grained Dataflow Specification for Fast HW/SW CoSynthesis", in *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004, pp. 24–29.

[JL98] C. Jäschke and R. Laur, "Resource Constrained Modulo Scheduling With Global Resource Sharing", in *Proceedings of the International Symposium on System Synthesis (ISSS)*, 1998, pp. 60–65.

[JMP+11] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing Hardware from Dataflow Programs", *Journal of Signal Processing Systems*, vol. 63, pp. 241–249, 2 2011.

[Joh75] D. B. Johnson, "Finding All the Elementary Circuits of a Directed Graph", *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.

[Kah74] G. Kahn, "The semantics of simple language for parallel programming", in *IFIP Congress*, 1974, pp. 471–475.

[KB06] V. Kianzad and S. S. Bhattacharyya, "Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 667–680, 2006.

[KBS14] M. A. B. Khadra, Y. Bai, and K. Schneider, "Synthesis of Distributed Synchronous Specifications to SysteMoC", in *Proceedings of the Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV)*, 2014, pp. 71–81.

[KDWV02] B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers, "A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach", in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2002, pp. 18–37.

[Kep93] D. Keppel, "Tools and techniques for building fast portable threads packages", University of Washington Department of Computer Science and Engineering, Tech. Rep., May 1993.

[Kil73] G. A. Kildall, "A Unified Approach to Global Program Optimization", in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, ACM, 1973, pp. 194–206.

[KKO+06] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "UML-based Multiprocessor SoC Design Framework", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 5, no. 2, pp. 281–320, May 2006.

[KMN+00] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. L. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform–Based Design", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2000.

[KR11] R. S. Khaligh and M. Radetzki, "A Metamodel and Semantics for Transaction Level Modeling", in *Proceedings of the Forum on Design Languages (FDL)*, 2011.

[KSS+09] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SYSTEMCODESIGNER - An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, pp. 1–23, 2009.

[KZP+07] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, "Parameterized Looped Schedules for Compact Representation of Execution Sequences in DSP Hardware and Software Implementation", in *IEEE Transactions on Signal Processing*, 2007, pp. 3126–3138.

[LA04] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar. 2004.

[LBBG86] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier, "SIGNAL–A Data Flow-Oriented Language for Signal Processing", *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34, no. 2, pp. 362–374, 1986.

[LBS+11] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky, "Implementing Domain-Specific Languages for Heterogeneous Parallel Computing", *IEEE Micro*, vol. 31, no. 5, pp. 42–53, 2011.

[Lee06] E. A. Lee, "The problem with threads", *IEEE Computer*, vol. 39, pp. 33–42, 2006.

[Lee89] E. A. Lee, "Scheduling strategies for multiprocessor real-time dsp", in *Proceedings of the Global Communications Conference, Exhibition & Industry Forum (GLOBECOM)*, 1989, pp. 1279–1283.

[Lee97] E. A. Lee, "A Denotational Semantics for Dataflow with Firing", EECS, University of California, Berkeley, CA, USA 94720, Tech. Rep., 1997.

[LF13] M. Lattuada and F. Ferrandi, "Modeling Pipelined Application with Synchronous Data Flow Graphs", in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2013, pp. 49–55.

[LKP+10] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha, "A Task Remapping Technique for Reliable Multi-Core Embedded Systems", in *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp. 307–316.

[LM87] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow", *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[LP95] E. A. Lee and T. Parks, "Dataflow Process Networks", in *Proceedings of the IEEE*, 1995, pp. 773–799.

[LS98] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[Mar06] G. Martin, "Overview of the MPSoC Design Challenge", in *Proceedings of the Design Automation Conference (DAC)*, 2006, pp. 274–279.

[MM08] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems", *Proceedings of the International Symposium on Computer Architecture (ISCA)*, vol. 36, no. 3, pp. 63–74, Jun. 2008.

[Mon14] Mono Project, *Mono*, `http://www.mono-project.com`, 2014.

[Mur89] T. Murata, "Petri nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[MVG+13] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An Overview of Today's High-Level Synthesis Tools", *Design Automation for Embedded Systems*, 2013.

[NEC14] NEC, *CyberWorkBench*, `http://www.nec.com/global/prod/cwb/`, 2014.

[NHUT07] B. Niemann, C. Haubelt, M. Uribe, and J. Teich, "Formalizing TLM with Communicating State Machines", in *Advances in Design and Specification Languages for Embedded Systems*, S. A. Huss, Ed., Springer, 2007, pp. 225–242.

[NS99] M. Nakamura and M. Silva, "Cycle Time Computation in Deterministically Timed Weighted Marked Graphs", in *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 2, 1999, pp. 1037–1046.

[NSD08] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, Mar. 2008.

[OGM+11] L. Ost, G. Guindani, F. Moraes, L. Indrusiak, and S. Määttä, "Exploring NoC-Based MPSoC Design Space with Power Estimation Models", *IEEE Design & Test of Computers*, vol. 28, no. 2, pp. 16–29, 2011.

[PDBB06] S. Pasricha, N. D. Dutt, E. Bozorgzadeh, and M. Ben-Romdhane, "Fabsyn: floorplan-aware bus architecture synthesis.", *IEEE Transactions on Very Large Scale Integrated Systems*, vol. 14, no. 3, pp. 241–253, 2006.

[PS04] H. D. Patel and S. K. Shukla, "Towards a Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models", in *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, 2004, pp. 248–253.

[PSB09a] W. Plishker, N. Sane, and S. S. Bhattacharyya, "Mode Grouping for More Effective Generalized Scheduling of Dynamic Dataflow Applications", in *Proceedings of the Design Automation Conference (DAC)*, Jul. 2009, pp. 923–926.

[PSB09b] W. Plishker, N. Sane, and S. Bhattacharyya, "A Generalized Scheduling Approach for Dynamic Dataflow Applications", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, Apr. 2009, pp. 111–116.

[PSK+08] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for Rapid Prototyping", in *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, 2008, pp. 17–23.

[Pto14] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[QP13] W. Quan and A. D. Pimentel, "A Scenario-Based Run-time Task Mapping Algorithm for MPSoCs", in *Proceedings of the Design Automation Conference (DAC)*, ACM, 2013, 131:1–131:6.

[RA04] D. L. Rosenband and Arvind, "Modular Scheduling of Guarded Atomic Actions", in *Proceedings of the Design Automation Conference (DAC)*, 2004, pp. 55–60.

[Sch09] K. Schneider, "The Synchronous Programming Language Quartz", Department of Computer Science, University of Kaiserslautern, Tech. Rep., Dec. 2009.

[Sch13] P. R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*, second edition. Springer, 2013.

[SDN06] T. Stefanov, E. Deprettere, and H. Nikolov, "Multi-Processor System Design with ESPAM", in *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006, pp. 211–216.

[SFH+06] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf, "Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, IEEE Computer Society, Mar. 2006, pp. 480–481.

[SGB+12] K. Sigdel, C. Galuzzi, K. Bertels, M. Thompson, and A. D. Pimentel, "Evaluation of Runtime Task Mapping Using the rSesame Framework", *International Journal of Reconfigurable Computing*, vol. 2012, 14:14–14:14, Jan. 2012.

[SGE+13] F. Siyoum, M. Geilen, J. Eker, C. von Platen, and H. Corporaal, "Automated Extraction of Scenario Sequences from Disciplined Dataflow Networks", in *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2013, pp. 47–56.

[SGHT09] M. Streubühr, J. Gladigau, C. Haubelt, and J. Teich, "Efficient Approximately-Timed Performance Modeling for Architectural Exploration of MPSoCs", in *Proceedings of the Forum on Design Languages (FDL)*, Sep. 2009.

[SGTB11] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-Aware Dataflow: Modeling, Analysis and Implementation of Dynamic Applications", in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2011, pp. 404–411.

[SJ04] I. Sander and A. Jantsch, "System Modeling and Transformational Design Refinement in ForSyDe", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, 2004.

[SLWS99] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Quasi-Static Scheduling of Embedded Software Using Equal Conflict Nets", in *Proceedings of the International Conference on Application and Theory of Petri Nets (ICATPN)*, Jun. 1999.

[SSL00] A. Sangiovanni-Vincentelli, M. Sgroi, and L. Lavagno, "Formal Models for Communication-based Design", in *Proceedings of the International Conference on Concurrency Theory (CONCUR)*, Aug. 2000.

[STG+01] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState - An Internal Design Representation for Codesign", *IEEE Transactions on Very Large Scale Integrated Systems*, vol. 9, pp. 558–565, 2001.

[SWN07] W. Sun, M. Wirthlin, and S. Neuendorffer, "FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 254–265, 2007.

[TCCS92] E. Teruel, P. Chrzastowski-Wachtel, J. M. Colom, and M. Silva, "On Weighted T-Systems", in *Proceedings of the International Conference on Application and Theory of Petri Nets (ICATPN)*, 1992, pp. 348–367.

[TGB+06] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk, "A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis", in *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2006, pp. 185–194.

[TH07] J. Teich and C. Haubelt, *Digitale Hardware/Software-Systeme: Synthese und Optimierung*, second edition. Springer, 2007.

[THH+11] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive Computing: An Overview", in *Multiprocessor System-on-Chip*, Springer, 2011, pp. 241–268.

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: a language for streaming applications", in *Proceedings of the International Conference on Compiler Construction (CC)*, ser. CC '02, Springer, 2002, pp. 179–196.

[TNS+07] M. Thompson, H. Nikolov, T. Stefanov, A. Pimentel, C. Erbas, S. Polstra, and E. Deprettere, "A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs", in *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 9–14.

[TSZ+99] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "FunState - An Internal Design Representation for Codesign", in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Nov. 1999, pp. 558–565.

[WL96] M. Williamson and E. Lee, "Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications", in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 1996, pp. 1340–1343.

[WM05] L. Wehmeyer and P. Marwedel, "Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, Mar. 2005, pp. 600–605.

[WR10] M. Wipliez and M. Raulet, "Classification and Transformation of Dynamic Dataflow Programs", in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2010, pp. 303–310.

[WR12] M. Wipliez and M. Raulet, "Classification of Dataflow Actors with Satisfiability and Abstract Interpretation", *International Journal of Embedded and Real-Time Communication Systems*, vol. 3, no. 1, pp. 49–69, 2012.

[WSS+11] H. H. Wu, C. C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A Model-Based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs", in *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS)*, 2011, pp. 66–77.

[WT85] R. A. Walker and D. E. Thomas, "A Model of Design Representation and Synthesis", in *Proceedings of the Design Automation Conference (DAC)*, 1985, pp. 453–459.

[Xil14] Xilinx, *Vivado High-Level Synthesis*, `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm`, 2014.

[ZGDS07] C. Zhu, Z. Gu, R. P. Dick, and L. Shang, "Reliable Multiprocessor System-on-Chip Synthesis", in *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 239–244.

Author's Own Publications

[FHZT13] J. Falk, C. Haubelt, C. Zebelein, and J. Teich, "Integrated Modeling Using Finite State Machines and Dataflow Graphs", in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., second edition, Springer, 2013, pp. 975–1013.

[FKH+10] J. Falk, J. Keinert, C. Haubelt, J. Teich, and C. Zebelein, "Integrated Modeling Using Finite State Machines and Dataflow Graphs", in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., first edition, Springer, 2010, pp. 1041–1075.

[FZH+10] J. Falk, C. Zebelein, C. Haubelt, J. Teich, and R. Dorsch, "Integrating Hardware/Firmware Verification Efforts Using SystemC High-Level Models", in *Proceedings of the Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV)*, Feb. 2010, pp. 137–146.

[FZHT11] J. Falk, C. Zebelein, C. Haubelt, and J. Teich, "A Rule-Based Static Dataflow Clustering Algorithm for Efficient Embedded Software Synthesis", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, Mar. 2011, pp. 521–526.

[FZHT13] J. Falk, C. Zebelein, C. Haubelt, and J. Teich, "A Rule-Based Quasi-Static Scheduling Approach for Static Islands in Dynamic Dataflow Graphs", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, 74:1–74:31, Apr. 2013.

[FZK+11] J. Falk, C. Zebelein, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya, "Analysis of SystemC Actor Networks for Efficient Synthesis", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 2, 18:1–18:34, Jan. 2011.

[HLMZ13] C. Haubelt, F. Ludwig, L. Middendorf, and C. Zebelein, "Using Stream Rewriting for Mapping and Scheduling Data Flow Graphs onto Many-Core Architectures", in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 2013, pp. 1431–1435.

[MZH13] L. Middendorf, C. Zebelein, and C. Haubelt, "Dynamic Task Mapping onto Multi-Core Architectures through Stream Rewriting", in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2013, pp. 196–204.

[ZFH+10] C. Zebelein, J. Falk, C. Haubelt, J. Teich, and R. Dorsch, "Efficient High-Level Modeling in the Networking Domain", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2010, pp. 1189–1194.

[ZFHT08] C. Zebelein, J. Falk, C. Haubelt, and J. Teich, "Classification of General Data Flow Actors into Known Models of Computation", in *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, Jun. 2008, pp. 119–128.

[ZHF+13] C. Zebelein, C. Haubelt, J. Falk, T. Schwarzer, and J. Teich, "Representing Mapping and Scheduling Decisions within Dataflow Graphs", in *Proceedings of the Forum on Design Languages (FDL)*, 2013, pp. 184–191.

[ZHF+14] C. Zebelein, C. Haubelt, J. Falk, T. Schwarzer, and J. Teich, "Model-Based Actor Multiplexing with Application to Complex Communication Protocols", in *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2014.

[ZHFT12a] C. Zebelein, C. Haubelt, J. Falk, and J. Teich, "A Model-Based Inter-Process Resource Sharing Approach for High-Level Synthesis of Dataflow Graphs", in *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn)*, 2012, pp. 17–22.

[ZHFT12b] C. Zebelein, C. Haubelt, J. Falk, and J. Teich, "Exploiting Model-Knowledge in High-Level Synthesis", in *Proceedings of the Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV)*, 2012, pp. 181–191.

[ZHFT13] C. Zebelein, C. Haubelt, J. Falk, and J. Teich, "Model-Based Representation of Schedules for Dataflow Graphs", in *Proceedings of the Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV)*, 2013, pp. 105–116.

Acronyms

ACFSM	Abstract Codesign Finite State Machine
AST	Abstract syntax tree
BDF	Boolean dataflow
BRAM	Block RAM
BSV	Bluespec SystemVerilog
BTH	Base transport header
CFDF	Core functional dataflow
CFG	Control flow graph
CFSM	Codesign Finite State Machine
CIL	Common Intermediate Language
CQ	Completion queue
CQE	Completion queue element
CSDF	Cyclo-static dataflow
DCT	Discrete cosine transform
DDF	Dynamic dataflow
DDN	Disciplined dataflow network
DE	Discrete-event
DFG	Dataflow graph
DFS	Depth-first search
DMA	Direct memory access
DSG	Dataflow schedule graph
DSM	Decision state modeling
ECFSM	Extended Codesign Finite State Machine
EIDF	Enable-invoke dataflow

EP	End-point
FCFS	First-come, first-served
FIFO	First in, first out
FPGA	Field-Programmable Gate Array
FPS	Frames per second
FSM	Finite state machine
FU	Functional unit
GST	Generalized schedule tree
HDF	Heterochronous dataflow
HLM	High-level model
HLS	High-level synthesis
HSDF	Homogeneous synchronous dataflow
IB	InfiniBand
IP	Intellectual property
ISA	Instruction set architecture
JPEG	Joint Photographic Experts Group
KPN	Kahn process network
LCA	Lowest common ancestor
LRU	Least recently used
LUT	Lookup table
MCU	Minimum coded unit
MG	Marked graph
MoC	Model of computation
MPSoC	Multiprocessor System-on-Chip
MTU	Maximum transmission unit
NoC	Network on chip
OSI	Open Systems Interconnection
PCIe	PCI Express
PSDF	Parameterized synchronous dataflow
PSOS	Periodic static order schedule
P/T	Place/transition

QCIF	Quarter common intermediate format
QoR	Quality of Results
QSS	Quasi-static schedule
RAM	Random-access memory
RAW	Read-after-write
RQ	Receive queue
RS	Reservation station
RSN	Reservation station number
RTE	Runtime environment
RTL	Register-transfer level
SADF	Scenario-aware dataflow
SANLP	Static affine nested loop program
SAS	Single-appearance schedule
SDF	Synchronous dataflow
SQ	Send queue
SR	Synchronous/reactive
TLM	Transaction level modeling
UML	Unified Modeling Language
WAR	Write-after-read
WAW	Write-after-write
WMG	Weighted marked graph
WQE	Work queue element

Theses

Model-Based Design Flow

- Embedded systems can be found in a wide range of applications like transportation systems, consumer electronics, medical equipment, industrial applications, or computer networking devices.
- Embedded systems have to meet a wide range of constraints like performance, power, area, reliability, safety, or security constraints. Besides these constraints, the time to market becomes more and more important.
- A design methodology is required which supports the automatic decision-making and refinement process at system level. Here, model-based design flows could be a solution.
- This thesis proposes a seamless model-based design flow from the system level to the instruction/logic level. To this end, it focuses on dataflow models, which can be used to implement a wide range of streaming applications. In a dataflow model, concurrent modules communicate and synchronize via packets transmitted over channels.

Model

- Modules (actors) are described by means of finite state machines (FSMs) whose transitions are based on guarded actions and exhibit a static communication behavior. These semantics provide for dynamic dataflow (DDF) actors, but still enable the identification of less expressive dataflow models.
- The sequential semantics of transitions provide for an efficient hardware/software synthesis of actors, and impose the same well-defined sequential semantics to hierarchical FSMs, which can be used to cope with the complexity of real-world applications.
- Transitions are based guarded actions. While it is shown that guard functions are not strictly required, they may greatly reduce the modeling complexity in certain cases.

Analysis

- The proposed dataflow model is expressive enough to model DDF applications. Thus, less expressive dataflow models must be identified in order to use domain-specific design methods developed for these models.
- For static dataflow models (HSDF, SDF, and CSDF), the proposed identification methodology has been successfully applied to a JPEG decoder in order to reduce the scheduling overhead imposed by the `IDCT2D` actor.
- For the BDF model, the proposed identification methodology has been extended to accommodate a data-dependent communication behavior.

System Synthesis

- Hierarchical actors are used to incorporate binding and scheduling decisions into the dataflow model. It is shown that a wide range of scheduling strategies can be represented by the proposed hierarchical model.
- In contrast to existing hierarchical dataflow models, hierarchical actors have the same dataflow semantics as non-hierarchical actors. In turn, the same analysis and optimization techniques can be applied to hierarchical and non-hierarchical actors.
- Child actors in the hierarchical model may be bound to different resources than their parent actor. Thus, appropriate binding-aware operational semantics are proposed and evaluated for the hierarchical model.

HW/SW Synthesis

- The hierarchical dataflow model at system level constitutes the input model to subsequent synthesis steps at the next lower levels of abstraction. In turn, this enables model-based optimizations at these lower levels of abstraction.
- A synthesis framework is introduced that permits, amongst others, the hardware/software synthesis of the proposed hierarchical dataflow model. To show the applicability of the framework, a JPEG decoder is synthesized to hardware and software.
- Applying model-based optimizations like token caching and the parallel evaluation of guards, the throughput of the JPEG decoder synthesized to hardware could be improved from 40 frames per second (FPS) to 100 FPS.
- In the context of inter-process resource sharing, a complex InfiniBand network adapter consisting of 2^{24} logical actor instances could be successfully multiplexed onto a single physical actor instance.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation mit dem Titel:

"A Model-Based Approach for the Specification and Refinement of Streaming Applications"

selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Arbeit wurde bisher weder im Ausland noch im Inland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Erlangen, 02.02.2015

_____

Christian Zebelein