SysteMoC: A Data-Flow Programming Language for Codesign

3

Joachim Falk, Christian Haubelt, Jürgen Teich, and Christian Zebelein

Abstract

Computations in hardware/software systems are inherently performed concurrently. Hence, modeling hardware/software systems requires notions of concurrency. Data-flow models have been and are still successfully applied in the modeling of hardware/software systems. In this chapter, we motivate and introduce the usage of data-flow models. Moreover, we discuss the expressiveness and analyzability of different data-flow Models of Computation (MoCs). Subsequently, we present SysteMoC, an approach supporting many data-flow MoCs based on the system description language SystemC. Besides specifying data-flow models, SystemMoC also permits the automatic classification of each different part of an application modeled in SysteMoC into a least expressive but most analyzable MoC. This classification is the key to further optimization in later design stages of hardware/software systems such as exploration of design alternatives as well as automatic code generation and hardware synthesis. Such optimization and refinement steps are employed as part of the SYSTEMCODESIGNER design flow that uses SysteMoC as its input language.

J. Falk (✉) • J. Teich
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
e-mail: joachim.falk@fau.de; juergen.teich@fau.de

C. Haubelt
Department of Computer Science and Electrical Engineering, Institute of Applied Microelectronics and Computer Engineering, University of Rostock, Rostock, Germany
e-mail: christian.haubelt@uni-rostock.de

C. Zebelein
Valeo Siemens eAutomotive Germany GmbH, Erlangen, Germany
e-mail: christian.zebelein.jv@valeo-siemens.com

© Springer Science+Business Media Dordrecht 2017
S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_4

Acronyms

BDF	Boolean Data Flow
CIC	Common Intermediate Code
CPU	Central Processing Unit
CSDF	Cyclo-Static Data Flow
DDF	Dennis Data Flow
DFG	Data-Flow Graph
DSE	Design Space Exploration
FIFO	First-In First-Out
FSM	Finite-State Machine
FunState	Functions Driven by State Machines
HSCD	Hardware/Software Codesign
HSDF	Homogeneous (Synchronous) Data Flow
KPN	Kahn Process Network
MoC	Model of Computation
NDF	Non-Determinate Data Flow
SDF	Synchronous Data Flow
SysteMoC	SystemC Models of Computation

Contents

3.1 Introduction

Due to the rising capabilities of embedded systems, their complexity has also increased tremendously. As a consequence, embedded systems are no longer implemented on a single computational resource, but in the form of a complex hardware/software system consisting of multiple connected heterogeneous resources including processor cores, hardware accelerators, and complex communication infrastructure to connect all these components. Hence, languages used for implementing and modeling applications to be mapped to such embedded systems need the ability to reflect and exploit the parallelism inherent in such target architectures.

Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.

— From "The Problem with Threads," by Edward A. Lee [27]

The above observation causes problems for the traditional implementation of embedded systems via sequential programming languages, as these languages typically handle concurrency only by using threads. *Data flow*, in contrast, is a modeling paradigm well-suited for the modeling of concurrent systems by concurrent *actors* that perform computation depending on the availability of *tokens* carrying data transmitted between them via First-In First-Out (FIFO) channels. Thus, data-flow models are particularly useful for modeling streaming applications as commonly found in the multimedia or networking domain and, hence, are a natural fit for modeling embedded systems, which should be implemented as codesigned hardware/software systems.

Over the last decades, many data-flow Models of Computation (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 data-flow MoC. It can be observed that *analyzability* of data-flow MoCs is inversely related to their expressiveness, i.e., there are problems which are decidable for less expressive data-flow MoCs, but are not decidable for more expressive ones (see Fig. 3.1). As analyzability of properties such as required bandwidth of channels, deadlock freedom, or schedulability issues is very important in early design phases of an embedded system, data-flow MoCs with a high analyzability are usually desirable. For example, scheduling of actors on computational resources 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 data-flow MoCs with limited expressiveness, as discussed below.

In Sect. 3.2, we give a survey and classify different data-flow models starting with *static* data-flow models and subsequently increasing the expressiveness of the models up to Non-Determinate Data Flow (NDF) [26]. Based on these discussions, we will introduce the *SystemC Models of Computation (SysteMoC) modeling language* [11,12] by example in Sect. 3.3. This language has strong formal underpinnings in data-flow modeling, but with the distinction that the expressiveness of the data-flow model used by an actor is not chosen in advance but determined from the implementation of the actor [12, 45]. Later on, in Sect. 3.4, we will provide a definition of the formal semantics of the language. To support modeling of even very complex real-world applications, the SysteMoC language realizes the NDF model. Nonetheless, to enable some SysteMoC applications to reap the benefits of analyzability of less expressive MoCs, the SysteMoC language—in

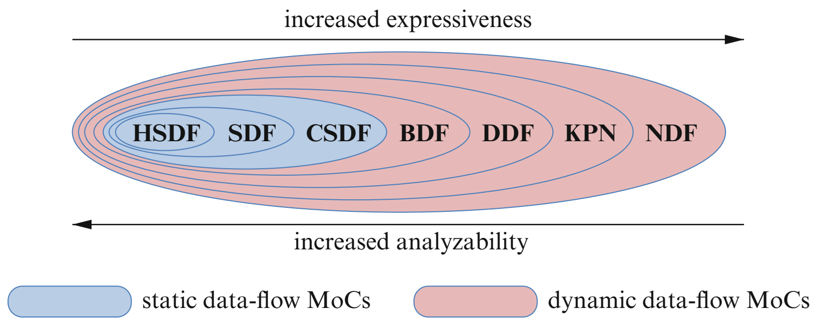


Fig. 3.1 Depicted above is a hierarchy [40] of various data-flow MoCs. The hierarchy is partitioned into static (*light blue*) data-flow MoCs and dynamic (*light red*) data-flow MoCs. A detailed explanation of the three static data-flow MoCs Homogeneous (Synchronous) Data Flow (HSDF), Synchronous Data Flow (SDF), and Cyclo-Static Data Flow (CSDF) that are well known from literature will be given in Sect. 3.2.2. Moreover, the dynamic data-flow MoCs Boolean Data Flow (BDF), Dennis Data Flow (DDF), Kahn Process Networks (KPNs), and Non-Determinate Data Flow (NDF) will be discussed in Sect. 3.2.3

contrast to general design languages such as SystemC—enforces a distinction between communication and computation of an actor. In Sect. 3.5, it will be shown how this distinction between communication and computation can be exploited in order to classify a SysteMoC actor into one of the static data-flow models (light blue in Fig. 3.1) in the hierarchy of data-flow model expressiveness. Hence, if the high expressiveness of SysteMoC is not used by a SysteMoC actor, then analysis techniques may detect this and classify the actor into a data-flow model of lower expressiveness but higher analyzability. The classification provides only a sufficient criterion if a general SysteMoC actor conforms to one of the static data-flow models. This limitation stems from the fact that the problem in general is undecidable. The chapter concludes with an overview of the SystemCoDesigner, a codesign framework based on SysteMoC as its design language. Here, we will give examples how SysteMoC may not only be used to integrate with Design Space Exploration (DSE) (More details on the DSE part within SystemCoDesigner can be found in ▸ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration".), but also subsequent hardware/software code generation in the refinement to its final implementation.

3.2 Overview of Basic Data-Flow Models

Data flow is a modeling paradigm well-suited for the modeling of concurrent systems by concurrently executing actors. Thus, data flow is a natural fit for the modeling of embedded hardware/software systems where the interaction and execution (control) of its functions is ruled by the availability of data. In the case of control-dominated systems, the *synchronous approach* presented in ▸ Chap. 2, "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems" is more convenient for their modeling. In the following, a general introduction to data flow is given in Sect. 3.2.1. Moreover, the trade-off between analyzability and expressiveness of these MoCs will be discussed. Data-flow models can be classified into *static* data-flow models, known for their high analyzability but low expressiveness, and *dynamic* data-flow models, known for their high expressiveness but low analyzability. Next, three important static data-flow models HSDF [6], SDF [28], and CSDF [3] known from literature will be recapitulated in Sect. 3.2.2. Finally, in Sect. 3.2.3, dynamic data-flow models such as KPNs [24] as well as usual realizations of them in the form of DDF [7] and BDF [5] are discussed.

3.2.1 Data Flow

A Data-Flow Graph (DFG) [24] is a graph consisting of vertices called *actors* and directed edges called *channels*. Whereas actors are used to model functionality, thus computations to be executed, channels represent data communication and storage requiring memory for their implementation. If not otherwise stated, channel memory is conceptually considered to be unbounded, thus representing a possibly infinite amount of storage. Moreover, the computation of an actor is usually [7] separated into distinct steps. These steps are called *actor firings*. An actor firing is an atomic computation step that consumes a number of data items called *tokens* from each incoming channel and produces a number of tokens on each outgoing channel. More formally, a DFG is defined as follows:

Definition 1 (Data-Flow Graph). A DFG is a directed graph $g = (A, C)$, where the set of vertices A represents the set of *actors* and the set of edges $C \subseteq A \times A$ represents the set of *channels*. Additionally, a *delay function* $\mathbf{delay} : C \rightarrow \mathscr{V}^*$ is given. (The "*"-operator is used to denote Kleene closure of a value set. It generates the set of all possible finite and infinite sequences of values from the value set, that is $X^* = \cup n \in \mathbb{N}_0 : X^n$. $\mathbb{N}_0$ denotes the set of non-negative integers, that is $\{0, 1, 2, \dots\}$.) It assigns to each channel $(a_{\text{src}}, a_{\text{snk}}) = c \in C$ a (possibly empty) sequence of initial tokens. (In some data-flow models that abstract from token values, the delay function may only return a non-negative integer that denotes the number of initial tokens on the channel. In such a context, the number of initial tokens may also be called the *delay* of a channel.) The set $\mathscr{V}$ is the set of data values which can be carried by a token. Finally, a channel capacity can be stated via the channel capacity function $\mathbf{size} : C \rightarrow \mathbb{N}_0$ that denotes the maximal number of tokens a channel can store.

An example of a very simple DFG according to Definition 1 is depicted in Fig. 3.2. It consists of two actors a_1 and a_2 which are connected by a channel c_1. A channel has, if not otherwise stated, an *infinite channel capacity*, i.e., it can store an infinite number of tokens that are in transit between the two actors connected by the channel. For notational convenience, the **src** and **snk** functions are used to refer, respectively, to the source actor, e.g., $\mathbf{src}(c_1) = a_1$, and the sink actor, e.g., $\mathbf{snk}(c_1) = a_2$, of a channel.

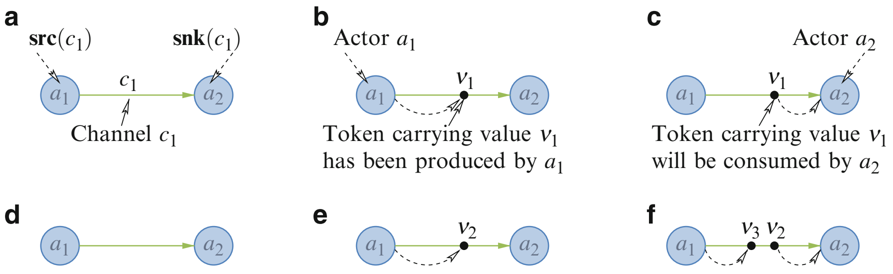

a
src(c_1)
snk(c_1)
c_1
a_1
a_2
Channel c_1
b
Actor a_1
v_1
Token carrying value v_1 has been produced by a_1
c
Actor a_2
Token carrying value v_1 will be consumed by a_2
d
e
v_2
f
v_3 v_2

Fig. 3.2 A DFG consisting of a single channel communicating data produced by actor a_1 and consumed by actor a_2. (**a**) Initial state of the DFG g. (**b**) Token production by actor a_1. (**c**) Token consumption by actor a_2. (**d**) Initial state is again reached after consumption of the token. (**e**) After production of a second token by actor a_1. (**f**) Production of a third token by a_1 while actor a_2 is consuming the second token

The channel c_1 enables a transmission of data values $\nu_1, \nu_2, \nu_3, \ldots$ from a_1 to a_2. Each data value is carried by a *token*. For data-flow models, a *token* represents the atomic unit of data production and consumption. Tokens are generally queued (as exemplified in Fig. 3.2b, e, f) and de-queued (Fig. 3.2c, f) on a channel in FIFO order. Between Fig. 3.2a, b, an *actor firing* of actor a_1 has occurred, producing a token with value ν_1. Next, in Fig. 3.2c, the first actor firing of actor a_2 consumes the token and its contained data value ν_1. The state of a DFG is given by the number and values of tokens on each channel as well as, possibly, the internal states of all actors. For analysis purposes of static data-flow models, the values of these tokens as well as the internal state of all actors may be ignored. Hence, a state equivalent to the initial state is again reached in Fig. 3.2d. The significance of this observation will be discussed in more detail in Sect. 3.2.2.1.

In Fig. 3.2d–f, another two tokens (ν_2 and ν_3) are produced by actor a_1 and the second token (ν_2) is in the process of being consumed by actor a_2. A resulting next state of the DFG after the second firing of actor a_2 is not depicted in Fig. 3.2, but consists of the DFG where only the token with value ν_3 remains on the channel.

An actor is *fireable* (also called *enabled*) if and only if all the tokens the next actor firing will consume are present on the input channels of the actor, e.g., the actor a_2 is enabled in Fig. 3.2b, c, e, f as a token, which is the only token that will be consumed by a firing of a_2, is present on the channel.

The literature on data-flow MoCs is very broad. Indeed, decades of research [1,3,5–7,11–14,16–19,21,22,24,28,29,31,33,34,41,45] into its applications have led to a multitude of different data-flow models. All of them make different trade-offs between their expressiveness and their analyzability, e.g., with respect to *deadlock freedom*, the ability to be executed in *bounded memory*, or the possibility to be *scheduled at compile time*. In the following, the most important data-flow MoCs are briefly reviewed, starting with those data-flow models with the least expressiveness.

3.2.2 Static Data Flow

Of primary interest for the expressiveness of a data-flow model are *production* and *consumption rates*. The *production rate* ($\mathbf{prod}(c)$) of an actor firing to a connected channel c is the number of tokens which are produced by that actor on the channel while firing. Consider Fig. 3.2b as an example where the first firing of actor a_1 produces one token onto channel c_1. Therefore, the production rate of the first firing of actor a_1 to channel c_1 is one. The *consumption rate* ($\mathbf{cons}(c)$) is defined analogously, e.g., Fig. 3.2c depicts a situation where the first firing of actor a_2 consumes one token from channel c_1. Thus, the consumption rate of the first firing of actor a_2 from channel c_1 is one. That is, the *consumption rate* of an actor firing from a connected channel is the number of tokens which are consumed by that actor from the channel while firing.

Now, an actor is called a *static data-flow actor* if its production and consumption rates are (1) *not dependent on the values of the tokens which are consumed by the actor*, (2) *not dependent on the points in time at which tokens arrive on the input channels or free places become available at the output channels*, and (3) *not dependent on some random process*. Thus, the communication behavior of a static actor can be fully predicted at compile time. The *consumption* and *production rates* of an actor are even further constrained in well-known static data-flow models, which are presented in the following.

3.2.2.1 Homogeneous Data Flow

The simplest data-flow model is Homogeneous (Synchronous) Data Flow (HSDF). (Note that the term *synchronous* in the name of two data-flow MoCs introduced in this chapter was coined in the original paper [29], but is, unfortunately, totally independent from the semantics of the term as used for *synchronous languages* that were introduced in ▶ Chap. 2, "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems".) Data-flow graphs corresponding to this data-flow model are also known as *marked graphs* [6] in literature. The *communication behavior* of HSDF actors is constrained in such a way that each actor firing must produce and consume exactly one token on, respectively, each outgoing and incoming channel, i.e., $\forall c \in C : \mathbf{cons}(c) = \mathbf{prod}(c) = 1$. Due to the low modeling power of the HSDF model, it is also highly analyzable; e.g., if each actor in an HSDF has fired exactly once, then the graph will be back to its initial state. If we assume that the DFG depicted in Fig. 3.2 is an HSDF graph, then firing actors a_1 and a_2 both once will transmit one token (ν_1) over the channel and lead back to the initial state (shown in Fig. 3.2a–d) where no token is present on the channel. These two actor firings $\langle a_1, a_2 \rangle$ realize a so-called *iteration* of the graph. It is proven in [6] that an HSDF graph has such an iteration if and only if each directed cycle of the graph contains at least one *initial token*. Briefly, a deadlock results in case there exists a cycle without any initial tokens in the graph because each actor that is part of this cycle can never fire as it awaits the production of a token by its predecessor actor in the cycle.

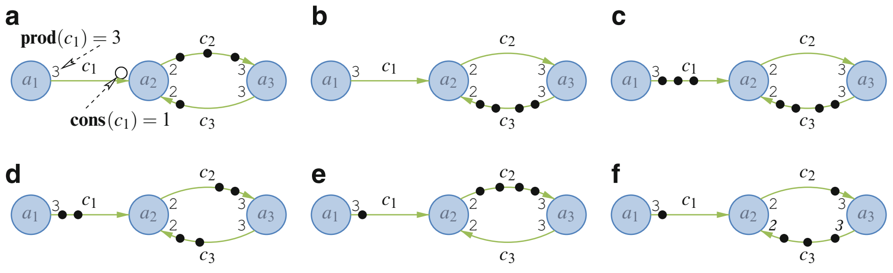


Fig. 3.3 Example of a Synchronous Data Flow graph and a sequence of actor firings realizing the iteration of the graph. (**a**) Initial state of the SDF graph. (**b**) After firing of actor a_3. (**c**) After firing of actor a_1. (**d**) After firing of actor a_2. (**e**) After firing of actor a_2. (**f**) After firing of actor a_3 (The next firing of actor a_2 will lead back to the initial state)

3.2.2.2 Synchronous Data Flow

In the Synchronous Data Flow (SDF) [29] MoC, the communication behavior of the actors is constrained to have consumption and production rates being *constant for all firings of an actor*. Moreover, consumption and production rates for all connected channels are assumed to be arbitrary positive integer constants. Therefore, in SDF, the consumption and production rates can be expressed by the *consumption rate function* **cons** : $C \to \mathbb{N}$ and the *production rate function* **prod** : $C \to \mathbb{N}$ that specify for each channel $c \in C$, respectively, the number of consumed tokens from the channel by an actor firing of the actor **snk**(c) and the number of produced tokens onto the channel by an actor firing of the actor **src**(c). (The symbol $\mathbb{N}$ is used to denote the set of natural numbers, that is the set $\{1, 2, 3, \ldots\}$.)

In visual representations of SDF graphs, as usual, the consumption and production rates are annotated at the beginnings and endings of the channel edges. An example of an SDF graph is depicted in Fig. 3.3a. Consumption and production rates of one, e.g., $\mathbf{cons}(c_1) = 1$, are traditionally not annotated to reduce clutter.

The question arises which conditions are *necessary and sufficient* for the existence of an iteration of a SDF graph. It is proven in [28] that such an iteration can be determined by *balance equations*. Each balance equation corresponds to one channel in the SDF graph. The balance equation for a channel c is given as: $\eta_{\mathbf{src}(c)} \cdot \mathbf{prod}(c) = \eta_{\mathbf{snk}(c)} \cdot \mathbf{cons}(c)$, where the variable $\eta_{\mathbf{src}(c)}$ denotes the number of actor firings of the actor producing tokens onto channel c while $\eta_{\mathbf{snk}(c)}$ denotes the number of actor firings of the actor consuming tokens from channel c. Given $\mathbf{prod}(c)$ and $\mathbf{cons}(c)$, the left and right sides of the equation denote the number of tokens that have been produced and consumed by the $\eta_{\mathbf{src}(c)}$ source actor and $\eta_{\mathbf{snk}(c)}$ sink actor firings, respectively. For the graph depicted in Fig. 3.3a, the balance equations corresponding to the three channels c_1 to c_3 of the graphs are as follows:

$$\eta_{a_1} \cdot 3 = \eta_{a_2} \cdot 1 \quad \eta_{a_2} \cdot 2 = \eta_{a_3} \cdot 3 \quad \eta_{a_3} \cdot 3 = \eta_{a_2} \cdot 2 \tag{3.1}$$

For an iteration, both sides of the equation must balance, otherwise an iteration would not bring the graph into the same state as it has started from. Hence, a solution besides the trivial zero solution is a *necessary* condition [28] for the existence of an iteration. If there exists such a solution for a static DFG, then this graph is called *consistent*.

In Equation (3.1), the number of firings for the actors a_1 to a_3 are given by η_{a_1} to η_{a_3}, respectively. Using the convention established in [2], the *minimum positive integer solution* to the set of balance equations, e.g., $\boldsymbol{\eta}^{\text{rep}} = (\eta_{a_1}, \eta_{a_2}, \eta_{a_3}) = (1, 3, 2)$, is called the *repetition vector* $\boldsymbol{\eta}^{\text{rep}}$ of an SDF graph. The length of this iteration is determined by summing all the entries of the repetition vector, e.g., for the SDF graph in Fig. 3.3a, the iteration can be realized by a sequence $\langle a_3, a_1, a_2, a_2, a_3, a_2\rangle$ of $\eta_{a_1} + \eta_{a_2} + \eta_{a_3} = 6$ actor firings as shown in Fig. 3.3a–f.

However, the existence of a repetition vector does not guarantee that a sequence of actor firings can be found that realizes the iteration. To exemplify, the four initial tokens of the SDF graph depicted in Fig. 3.3a are removed. This does not change the calculation or existence of the *repetition vector* of the SDF graph. However, without any initial tokens, neither actor a_2 nor actor a_3 can ever be fired. In general, a *necessary and sufficient criterion of the existence of the iteration* is to test whether a computed repetition vector may also execute as an iterative deadlock-free schedule by firing each fireable actor as many times as implied by the repetition vector until the iteration is finished or a deadlock has occurred. Note that in the general case, the length of the iteration may be exponential in the size of the SDF graph. Hence, in contrast to HSDF models, where the question of the existence of an iteration can be answered in polynomial time [20], the problem is only solvable in exponential time for SDF graphs [32].

3.2.2.3 Cyclo-Static Data Flow

An extension of the SDF model is Cyclo-Static Data Flow (CSDF). In the CSDF [3] MoC, the communication behavior of an actor is extended to allow for cyclically varying consumption and production rates between actor firings. The length of this cycle is known as the number τ of *phases* of a CSDF actor. An actor firing of a CSDF actor is also known as a CSDF *phase*. To accommodate the cyclically varying consumption and production rates, the consumption and production rate functions have to be extended to return vectors of length τ (the number of phases of the CSDF actor consuming or producing tokens), i.e., the functions $\textbf{cons} : C \to \mathbb{N}_0^\tau$ and $\textbf{prod} : C \to \mathbb{N}_0^\tau$ specify for each channel $c \in C$ a vector $(n_0, n_1, \ldots, n_{\tau-1})$ where each vector entry corresponds to the consumption or production rate of the CSDF actor in the corresponding phase.

The question whether there exists an iteration can again be answered by solving the appropriate balance equations—answering the question of the consistency of the CSDF graph—and if the graph is consistent by testing whether the computed repetition vector may also execute as an iterative deadlock-free schedule. For the purpose of calculating the repetition vector, all CSDF actors can be replaced by SDF actors with consumption and production rates derived by summing all rates in the corresponding vectors of consumption and production rates of the CSDF actors.

Hence, the balance equation for a channel c from actor a_{src} to actor a_{snk} with the respective production and consumption rates $\mathbf{prod}(c) = (n_0, n_1, \ldots n_{\tau_{\text{src}}-1})$ and $\mathbf{cons}(c) = (m_0, m_1, \ldots m_{\tau_{\text{snk}}-1})$ is as follows:

$$\frac{\eta_{a_{\text{src}}}}{\tau_{\text{src}}} \cdot (n_0 + n_1 + \ldots + n_{\tau_{\text{src}}-1}) = \frac{\eta_{a_{\text{snk}}}}{\tau_{\text{snk}}} \cdot (m_0 + m_1 + \ldots + m_{\tau_{\text{snk}}-1})$$

As is the case for SDF models, the smallest positive integer solution of the *balance equations* uniquely determines the (minimal) repetition vector of the CSDF graph. Finally, the existence of a valid iteration corresponding to the repetition vector needs to be verified as—like in SDF—the existence of the repetition vector is only a necessary but not sufficient criterion for the existence of the iteration.

3.2.3 Dynamic Data Flow

In contrast to *static data-flow models*, where the consumption and production rates cannot be influenced by the values of the consumed tokens, dynamic data-flow actors can vary their consumption and production rates depending on the *history of the consumed tokens* and also *on the tokens to be consumed*. Dynamic data flow is the first introduced data-flow model where consumption and production rates depend on the data values being consumed and produced.

3.2.3.1 Boolean Data Flow

Boolean Data Flow (BDF) [5] can be seen as an extension of the class of static data-flow models by introducing two dynamic actor types, the *switch* actor and the *select* actor. In the following, the notion of ports will be used interchangeably with the channels connected to these ports. Hence, expressions like $\mathbf{cons}(i)$ and $\mathbf{prod}(o)$ are used to refer to the consumption rate and production rate on the channel connected to the respective input or output port. This enables us to depict an actor and show its implementation in isolation, e.g., as has been used in Fig. 3.4.

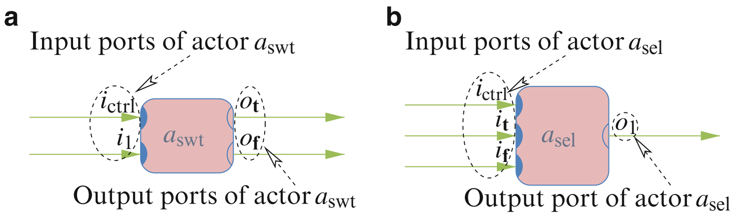

a
Input ports of actor a_{swt}
i_{ctrl}
i_1
a_{swt}
o_{t}
o_{f}
Output ports of actor a_{swt}
b
Input ports of actor a_{sel}
i_{ctrl}
i_{t}
i_{f}
a_{sel}
o_1
Output port of actor a_{sel}

Fig. 3.4 Shown here are the BDF switch and select actors with their corresponding input and output ports. The color scheme chosen to distinguish static and dynamic actors has been selected according to the colors marking static and dynamic MoCs in Fig. 3.1. (**a**) Depiction of the switch actor a_{swt} in isolation. (**b**) Depiction of the select actor a_{sel} in isolation

Moreover, let **t** (true) and **f** (false) denote the Boolean truth values. The *switch* actor, depending on the truth value of a control token (see Fig. 3.4a) from its control input port i_{ctrl}, forwards a token from its input port i_1 to either its true ($o_{\mathbf{t}}$) or its false ($o_{\mathbf{f}}$) output port. The *select* actor acts in the opposite way, i.e., depending on the truth value of a control token (see Fig. 3.4b) from its control input port i_{ctrl}, it forwards a token from either its true ($i_{\mathbf{t}}$) or its false ($i_{\mathbf{f}}$) input port to its output port o_1.

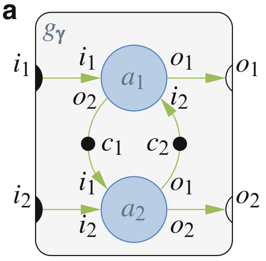
The usage of these two dynamic actor types together with the arithmetic primitives enable the construction of arbitrary control flow structures. The channels (of infinite capacity) can be used to represent an infinite memory. Simple arithmetic operations are supported by BDF via its ability to model static actors, e.g., like an SDF actor implementing an addition. Together with the control flow logic, a Turing machine can be implemented by the BDF model [5]. Hence, the existence of iterations or the problem of execution in bounded memory is in general already undecidable for BDF graphs.

3.2.3.2 Dennis Data Flow

Dennis Data Flow (DDF) [7] is an extension of BDF by allowing all dynamic actors that are realizable by using *(blocking) read* and *(blocking) write* communication primitives. Code inside DDF actors is assumed to be executed sequentially, e.g., as seen in Fig. 3.5b, c. A blocking read or write primitive is used to receive (see Lines 3 to 4) or transmit (see Lines 5 to 6) one data value on an input or output channel, respectively. Once a blocking read or write primitive is invoked, the execution of the actor will block until the data value has been successfully received or transmitted. Hence, at most one blocking read or write primitive can be active at one point in time.

All MoCs more general than BDF according to Fig. 3.1 are Turing complete. However, one aspect of difference is the modeling power of a single actor. An

a



A data-flow subgraph consisting of two HSDF actors

b

```
1: procedure $a_1$
2:     while t do
3:         $v_1 \leftarrow \mathrm{read}(a_1.i_1)$
4:         $v_2 \leftarrow \mathrm{read}(a_1.i_2)$
5:         $\mathrm{write}(a_1.o_1, v_1)$
6:         $\mathrm{write}(a_1.o_2, v_2)$
7:     end while
8: end procedure
```

Implementation of the HSDF actor a_1 via the read and write operations provided by DDF

c

```
1: procedure $a_2$
2:     while t do
3:         $v_1 \leftarrow \mathrm{read}(a_2.i_1)$
4:         $v_2 \leftarrow \mathrm{read}(a_2.i_2)$
5:         $\mathrm{write}(a_2.o_1, v_1)$
6:         $\mathrm{write}(a_2.o_2, v_2)$
7:     end while
8: end procedure
```

Implementation of the HSDF actor a_2 via the read and write operations provided by DDF

Fig. 3.5 Depicted above (see (**a**)) is the subgraph g_γ that can be connected with input and output ports i_1, i_2, o_1, and o_2 to an unspecified environment. The subgraph consists of two DDF actors a_1 (see (**b**)) and a_2 (see (**c**)) that both implement a communication behavior that corresponds to the HSDF MoC

important question here is whether it is possible to take an arbitrary connected subgraph of a DFG, e.g., the one shown in Fig. 3.5a, and represent it as an actor in the data-flow model. This property is called *compositionality* of the MoC. If a data-flow model is compositional, then the highly desirable operation of abstraction becomes seamlessly possible. An abstraction operation can be performed by hiding the implementation complexity of an arbitrary connected subgraph of the model behind the interface of an actor. If the data-flow model is compositional, then this *composite actor*, which represents the subgraph, can be handled like any other actor in the system. Otherwise, the composite actor, e.g., the actor a_γ that represents the subgraph g_γ, is always an exception and needs special treatment.

To exemplify, we consider the question of the least expressive MoC required to express the composite actor a_γ for the subgraph g_γ. All HSDF actors can also be represented by DDF actors, e.g., as shown in Fig. 3.5b, c for the actors a_1 and a_2 of the subgraph g_γ. In contrast, due to the constraint that DDF actors can only use (blocking) read to access data from their input ports, there is no composite actor a_γ that corresponds to the DDF MoC. Here, the problematic situation is that the composite actor a_γ has to first produce a token on either output port o_1 or output port o_2 depending on whether a token arrives first at either input port i_1 or input port i_2, respectively. However, due to the (blocking) read semantics of DDF, there is no possibility to detect on which input port a token arrives first. Once an input port has been chosen for a read access, the DDF actor cannot process any tokens from any other input port until a token has been successfully read.

Moreover, as the MoCs HSDF, SDF, CSDF, and BDF are all less expressive than DDF, there is also no composite actor a_γ that corresponds to one of these less expressive MoCs. Hence, as the actors a_1 and a_2 are of the least expressive MoC HSDF and the composite actor cannot be expressed via the DDF MoC, all MoCs up to and including DDF are non-compositional. In contrast, as will be shown in the next section, the composite actor a_γ can be expressed as the Kahn function κ_{a_γ}.

One could argue that the expressiveness of a DDF actor is not a proper superset of the expressiveness of an HSDF actor due to the issue of atomicity in the consumption and production of tokens that is implied by the notion of firing of an HSDF actor. In contrast, read or write communication primitives used by DDF actors consume or produce tokens in isolation, thus allowing the firing of other actors to interrupt a sequence of read or write communication primitives in the DDF actor. However, if we consider DFGs containing only actors of the expressiveness KPN and below, then the issue of atomicity is not a relevant criterion for the functionality of a DFG due to the *sequence determinate* [30] nature of such DFGs. Briefly, if a DFG is sequence determinate, then the behavior of the DFG is independent from the sequence of actor firings that are taken to schedule the graph. Thus, if a sequence of read or write communication primitives is interrupted by other actor firings or not does not influence the functionality of the DFG. In conclusion, the issue of atomicity is not relevant for the proof of non-compositionality of MoCs DDF and below.

If atomicity should also be considered in the hierarchy of data-flow model expressiveness, then the semantics of DDF must be extended to allow grouping of sequences of read or write communication primitives to be executed atomically or not at all. Unsurprisingly, such an extension of the semantics of DDF still does not allow a DDF actor to model the data-flow subgraph g_γ from Fig. 3.5a.

3.2.3.3 Kahn Process Networks

Kahn Process Networks (KPNs) are one of the oldest data-flow MoCs. The original paper [24] of Kahn used a *denotational semantics* to describe the behavior of a KPN actor. In this denotational semantics, an actor a is described by a Kahn function κ_a. To exemplify, the two (identical) HSDF actors a_1 and a_2 in Fig. 3.5a are represented by the two (identical) Kahn functions κ_{a_1} and κ_{a_2} given in Equation (3.2).

$$\kappa_{a_1}(s_x, s_y) = \kappa_{a_2}(s_x, s_y) = \begin{cases} (\langle\rangle, \langle\rangle) & \text{if } \#s_x = 0 \vee \#s_y = 0 \\ (\langle \mathbf{hd}(s_x)\rangle, \langle \mathbf{hd}(s_y)\rangle) \overset{\rightharpoonup}{\frown} \kappa_{a_1}(\mathbf{tl}(s_x), \mathbf{tl}(s_y)) & \text{otherwise} \end{cases} \tag{3.2}$$

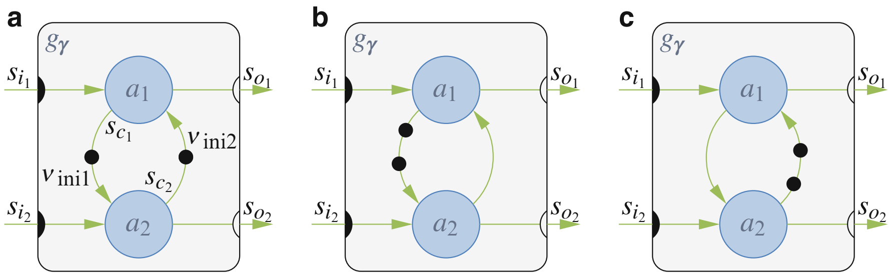
Here, a so-called *signal* $s \in S \equiv \mathscr{V}^*$ is (a possibly infinite) sequence of values carried by the tokens being transported over a channel, e.g., $\langle 5, 8, 7\rangle$ for a sequence of three values. *The length of a signal*, i.e., the number of values contained in it, will be denoted by the $\#s$ notation, e.g., $\#\langle 5, 8, 7\rangle = 3$. The $\mathbf{hd}(s)$ and the $\mathbf{tl}(s)$ notations are used to access the *head* of a sequence and, respectively, the *tail* of a sequence, i.e., the sequence without its head.

Considering Equation (3.2), we see that the Kahn function returns a tuple of empty sequences $(\langle\rangle, \langle\rangle)$ if at least one of the input signals s_x or s_y is empty, i.e., their length is zero. Otherwise, the Kahn function is called recursively with the tails of both input sequences, i.e., $\kappa_{a_1}(\mathbf{tl}(s_x), \mathbf{tl}(s_y))$, and the result of this computation is concatenated to the tuple of single value sequences containing the head of both input sequences, i.e., $(\langle \mathbf{hd}(s_x)\rangle, \langle \mathbf{hd}(s_y)\rangle)$. To demonstrate, we perform the following Kahn function application:

$$\begin{aligned}
\kappa_{a_1}(\langle 5,8,7\rangle, \langle 9,8\rangle) &= (\langle \mathbf{hd}(\langle 5,8,7\rangle)\rangle, \langle \mathbf{hd}(\langle 9,8\rangle)\rangle) \overset{\rightharpoonup}{\frown} \kappa_{a_1}(\mathbf{tl}(\langle 5,8,7\rangle), \mathbf{tl}(\langle 9,8\rangle)) \\
&= (\langle 5\rangle, \langle 9\rangle) \overset{\rightharpoonup}{\frown} \kappa_{a_1}(\langle 8,7\rangle, \langle 8\rangle) = (\langle 5\rangle, \langle 9\rangle) \overset{\rightharpoonup}{\frown} (\langle 8\rangle, \langle 8\rangle) \overset{\rightharpoonup}{\frown} \kappa_{a_1}(\langle 7\rangle, \langle\rangle) \\
&= (\langle 5\rangle, \langle 9\rangle) \overset{\rightharpoonup}{\frown} (\langle 8\rangle, \langle 8\rangle) \overset{\rightharpoonup}{\frown} (\langle\rangle, \langle\rangle) = (\langle 5\rangle \frown \langle 8\rangle \frown \langle\rangle, \langle 9\rangle \frown \langle 8\rangle \frown \langle\rangle) \\
&= (\langle 5,8\rangle, \langle 9,8\rangle)
\end{aligned}$$

Here, the "$\frown$"-operator is used to concatenate two sequences, and the "$\overset{\rightharpoonup}{\frown}$"-operator is applied to tuples of signals by pointwise extension of the "$\frown$"-operator.

In general, a Kahn function $\kappa_a : S^n \to S^m$ transforms n value sequences on its n input ports to m value sequences on its m output ports. All Kahn functions are required [30] to be Scott-continuous. In practice, this means that appending values to the input sequences of the Kahn function can only result in appending values to the resulting output sequences, i.e., if $\kappa_a(s_{i_1}, s_{i_2}, \ldots s_{i_n}) = (s_{o_1}, s_{o_2}, \ldots s_{o_m})$, then $\kappa_a(s_{i_1} \frown s'_{i_1}, s_{i_2} \frown s'_{i_2}, \ldots s_{i_n} \frown s'_{i_n}) = (s_{o_1} \frown s'_{o_1}, s_{o_2} \frown s'_{o_2}, \ldots s_{o_m} \frown s'_{o_m})$.



Start state in which Kahn function κ_{a_γ} is active

State in which the helper function κ_{h1} is active

State in which the helper function κ_{h2} is active

Fig. 3.6 Above (see (**a**)), the subgraph g_γ from Fig. 3.5 is depicted with the corresponding annotations to express it as a partial KPN. Moreover, in (**b**) and (**c**) the internal states encountered in the application of Kahn function κ_{a_γ} are illustrated

The behavior of a KPN is given by the *least fixed point* of an equation system that represents the connections of these actors to each other, e.g., as given in Equations (3.3) and (3.4) for the topology shown in Fig. 3.6a. The connection between actors a_1 and a_2 is provided by the signal s_{c_1} (see Equation (3.3)) produced by actor a_1 and the signal s_{c_2} (see Equation (3.4)) produced by actor a_2. The initial tokens ν_{ini1} and ν_{ini2} on the channels c_1 and c_2 connecting these two actors are modeled by prepending the corresponding initial token value in front of the corresponding signal, e.g., $\langle \nu_{\text{ini1}} \rangle ^\frown s_{c_1}$ prepends the value ν_{ini1} in front of the values carried by the signal s_{c_1}.

$$(s_{o_1}, s_{c_1}) = \kappa_{a_1}(s_{i_1}, \langle \nu_{\text{ini2}} \rangle ^\frown s_{c_2}) \tag{3.3}$$

$$(s_{o_2}, s_{c_2}) = \kappa_{a_2}(s_{i_2}, \langle \nu_{\text{ini1}} \rangle ^\frown s_{c_1}) \tag{3.4}$$

It turns out that such an equation system, e.g., Equations (3.2), (3.3), and (3.4), has a least fixed point for any input signal [30], and the function mapping the input signal to the corresponding fixed point solution of the equation system again represents a Kahn function. In that sense, the KPN model—in contrast to DDF and all MoCs of lower expressive power—is compositional. Another important characteristic emerges from this fact. The behavior of a *KPN model and all MoCs of lower expressive power*, be it a complete graph or a subgraph, is independent from the scheduling of actors, which is given by the operational implementation. Such data-flow models are called *sequence determinate* [30]. However, due to the non-compositionality of DDF, the resulting Kahn function is not generally representable via the operational semantics of DDF.

To exemplify, the Kahn function κ_{a_γ}—that is, the expression of the composite actor a_γ as a Kahn function—for the least fixed point of the Equations (3.2), (3.3), and (3.4) is given below. This function is defined via the main Kahn function κ_{a_γ} (given in Equation (3.5)) and the two helper functions κ_{h1} and

κ_{h2} (given in Equations (3.6) and (3.7)) that recursively call each other. The main Kahn function κ_{a_γ} is active in the state shown in Fig. 3.6a and can call either helper function κ_{h1} or κ_{h2} depending on whether a token is present at the head of either the input signal s_{i_1} or the input signal s_{i_2}, respectively. This will lead to the states depicted in Fig. 3.6b, c. From these states, via a call to the main Kahn function κ_{a_γ}, a transition back to the start state is performed if a token is present at the head of input signal s_{i_2} (for helper function κ_{h1}) or input signal s_{i_1} (for helper function κ_{h2}).

$$\kappa_{a_\gamma}(s_{i_1}, s_{i_2}) = \begin{cases} (\langle \mathbf{hd}(s_{i_1}) \rangle, \langle\rangle) \overset{\rightarrow}{\frown} \kappa_{\mathrm{h1}}(\mathbf{tl}(s_{i_1}), s_{i_2}) & \text{if } \#s_{i_1} \geq 1 \\ (\langle\rangle, \langle \mathbf{hd}(s_{i_2}) \rangle) \overset{\rightarrow}{\frown} \kappa_{\mathrm{h2}}(s_{i_1}, \mathbf{tl}(s_{i_2})) & \text{if } \#s_{i_2} \geq 1 \\ (\langle\rangle, \langle\rangle) & \text{otherwise} \end{cases} \tag{3.5}$$

$$\text{where } \kappa_{\mathrm{h1}}(s_{i_1}, s_{i_2}) = \begin{cases} (\langle\rangle, \langle \mathbf{hd}(s_{i_2}) \rangle) \overset{\rightarrow}{\frown} \kappa_{a_\gamma}(s_{i_1}, \mathbf{tl}(s_{i_2})) & \text{if } \#s_{i_2} \geq 1 \\ (\langle\rangle, \langle\rangle) & \text{otherwise} \end{cases} \tag{3.6}$$

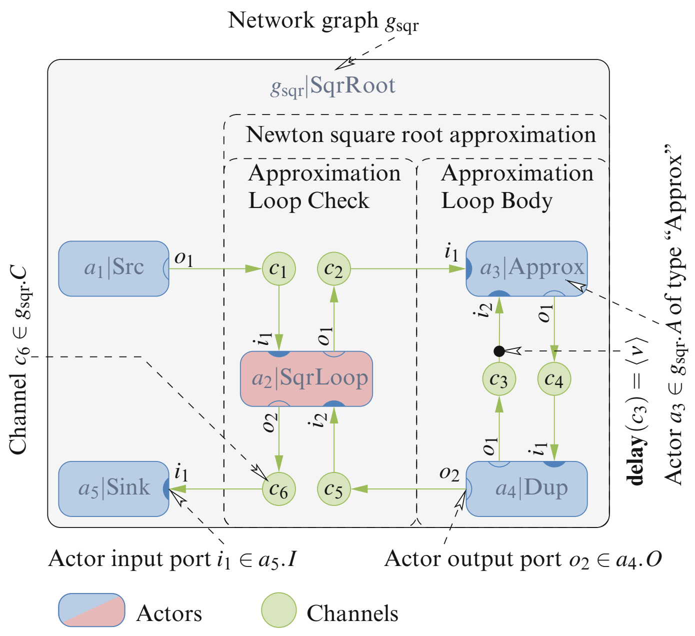
$$\text{where } \kappa_{\mathrm{h2}}(s_{i_1}, s_{i_2}) = \begin{cases} (\langle \mathbf{hd}(s_{i_1}) \rangle, \langle\rangle) \overset{\rightarrow}{\frown} \kappa_{a_\gamma}(\mathbf{tl}(s_{i_1}), s_{i_2}) & \text{if } \#s_{i_1} \geq 1 \\ (\langle\rangle, \langle\rangle) & \text{otherwise} \end{cases} \tag{3.7}$$

Data-flow models with higher expressiveness are of the Non-Determinate Data Flow (NDF) MoC. This model as well as all previously discussed data-flow MoCs may be specified in the SystemC-based actor-oriented language SysteMoC that will be introduced in the next section.

3.3 Informal Introduction to SysteMoC

In this section, the SysteMoC modeling language [11, 12], a class library based on SystemC, will be presented. In SysteMoC parlance, data-flow graphs are called *network graphs*. As a running example, a network graph (see Fig. 3.7) of an application implementing Newton's iterative square root algorithm will be used throughout this section. Network graphs are very similar to DFGs as introduced in Definition 1, but are bipartite graphs consisting of channels $c \in C$ and actors $a \in A$. These vertices are connected via point-to-point connections between a channel and either an actor input or an output port. This acknowledges the fact that actors and channels must both be realized by some kind of resource and, hence, during DSE [25] a binding of vertices to resources of an architecture has to be explored. The *network graph* g_{sqr} implements Newton's iterative algorithm for calculating the square roots of an infinite input sequence generated by the `Src` actor a_1. Here, the square root values are computed by Newton's iterative algorithm realized via the `SqrLoop` actor a_2 performing error bound checking and the actors a_3 - a_4 performing an approximation step. After satisfying the error bound, the result is transported to the `Sink` actor a_5.

In the following, we will learn how to realize a network graph by instantiating actors and FIFO channels as well as connecting these FIFO channels with the ports



Network graph g_{sqr}
g_{sqr}|SqrRoot
Newton square root approximation
Approximation Loop Check
Approximation Loop Body
a_1|Src
a_2|SqrLoop
a_3|Approx
a_4|Dup
a_5|Sink
c_1
c_2
c_3
c_4
c_5
c_6
o_1
o_2
i_1
i_2
Channel $c_6 \in g_{\text{sqr}}.C$
$\mathbf{delay}(c_3) = \langle v \rangle$
Actor $a_3 \in g_{\text{sqr}}.A$ of type "Approx"
Actor input port $i_1 \in a_5.I$
Actor output port $o_2 \in a_4.O$
Actors
Channels

Fig. 3.7 The *network graph* g_{sqr} [10] displayed above implements Newton's iterative algorithm for calculating square roots. Actors are *bordered dark blue* and *shaded* according to their MoC (see Fig. 3.1 on page 62), while channels are depicted in *green*

of the actors. Next, in Sect. 3.3.2, we will study how to write the actor classes that are instantiated in the previous step. Finally, in Sect. 3.3.3, it will be shown how to specify the consumption and production rates exhibited by these actors via usage of so-called *actor Finite-State Machines (FSMs)*.

3.3.1 Specification of the Network Graph

SysteMoC is an open source C++ class library. Hence, all components mentioned in the previous section are represented by C++ classes. As an example, we will specify the `SqrRoot` C++ class that corresponds to the g_{sqr} DFG from Fig. 3.7 in Listing 1. All C++ classes representing DFGs must be derived from the `smoc_graph` base class provided by the SysteMoC library, e.g., as is shown in Line 1. All actors of a DFG must be instantiated in the constructor of the corresponding class, e.g., the lines colored dark blue as is shown in Lines 3 to 7 declaring the actor member variables that are instantiated in the constructor in Line 11. Subsequently, in the body of the constructor, e.g., Lines 12 to 20, which are colored green, the FIFO channels $c_1, c_2 \ldots c_6$ are connected to the input and output ports of these actors via the `connectNodePorts` method of SysteMoC.

In the simplest case (Lines 12 to 15), the `connectNodePorts` method takes two arguments: the output port o, from where the channel starts, and the input

Listing 1 Corresponding `SqrRoot` [10] class for the network graph g_{sqr}

```
 1 class SqrRoot: public smoc_graph {
 2 protected:
 3   Src      a1;
 4   SqrLoop  a2;
 5   Approx   a3;
 6   Dup      a4;
 7   Sink     a5;
 8 public:
 9   SqrRoot(sc_module_name name)
10     : smoc_graph(name),
11       a1("a1"), a2("a2"), a3("a3"), a4("a4"), a5("a5") {
12     connectNodePorts(a1.o1, a2.i1); // $c_1$
13     connectNodePorts(a2.o1, a3.i1); // $c_2$
14     connectNodePorts(a2.o2, a5.i1); // $c_6$
15     connectNodePorts(a4.o2, a2.i2); // $c_5$
16     connectNodePorts(a3.o1, a4.i1,  // $c_4$
17       smoc_fifo<double>(2));        // size($c_4$) = 2
18     connectNodePorts(a4.o1, a3.i2,  // $c_3$
19       smoc_fifo<double>(3)          // size($c_3$) = 3
20         << 2.0);                    // delay($c_3$) = ⟨2.0⟩
21   }
22 };
```

port i, which is the destination of the channel. However, it is also possible to explicitly parameterize the created channel c with its channel capacity of **size**(c) tokens and a sequence of initial tokens **delay**(c). To exemplify, consider Lines 17 and 19, where a third parameter, the *channel initializer* `smoc_fifo<T>(n) « initial tokens...`, is given to the method `connectNodePorts`. If no channel initializer is given, then a FIFO channel with a channel capacity of one token and without any initial tokens is created between the output port o and the input port i. If a channel initializer is given, then it must be parameterized with the data type `T` carried by the channel and the channel capacity `n` in units of tokens. If initial tokens are required, these can be provided to the channel initializer via a sequence of "«"-operators each followed by an initial token, e.g., $\ll \nu_1 \ll \nu_2 \ll \ldots \nu_n$. Consider Line 17 as an example of how to specify a FIFO channel with a channel capacity of two tokens. In this case, the channel initializer is parameterized with the C++ `double` data type, denoting that the token values carried by the channel will be of the C++ `double` data type. An example for providing an initial token is given in Line 20, where the "«"-operator is used to provide the `double` value 2.0 as an initial token for the created channel between the ports $a_4.o_1$ and $a_3.i_2$.

3.3.2 Specification of Actors

Each actor in SysteMoC is represented by a C++ class, e.g., the class `SqrLoop` as defined in the following Listing 2 is representing the actor a_2 shown in Fig. 3.7, that is derived from the `smoc_actor` base class (see Listing 2 Line 1) provided by the SysteMoC library. The input and output ports of an actor are specified by member variables of type `smoc_port_in` and `smoc_port_out` as exemplified in Listing 2 Lines 3 and 4, respectively. (Standard SystemC FIFO ports could not be used as the semantics of SysteMoC FIFOs extends standard FIFO semantics by the concept of a *random-access region* as shown in Fig. 3.9.) Furthermore, actors can have member variables, e.g., the variable `v` declared in Line 6. The functionality of an actor is represented by methods of the class, e.g., for the `SqrLoop` actor, the Lines 8 to 11 in Listing 2. Moreover, these methods are distinguished into *actions* (colored cyan), which can modify member variables, and *guards* (colored brown), which are declared as `const` methods and, hence, are not allowed to update the member variables. Later, in Sect. 3.3.3, it will be discussed how these action and guard methods are used as part of the actor FSM.

The `copyStore` method is responsible for forwarding the input token value on input port i_1 to the output port o_1 and storing the value into the member variable `v` for later replication on the output port o_1 by the method `copyInput`. This replication is required if the achieved accuracy by one square root iteration step by actor a_3 is below the bound determined by the guard `check`. On the other hand, if the approximation is within the error bound, then the action `copyApprox` is executed to forward this approximation to the output port o_2.

Note that actions themselves do not control token production or consumption, but only read or write values of input or output tokens via the syntax `i[n]` and `o[m]` that is used in Listing 2 and Fig. 3.9 to access the `n`th token relative to the read pointer, respectively, the `m`th token relative to the write pointer of the ring buffer (see Fig. 3.9) that realizes the channel that is connected to the corresponding port. The communication behavior, i.e., token production and consumption, is controlled solely by the actor FSM as will be detailed next.

Listing 2 SysteMoC implementation of the SqrLoop actor a_2

```
 1 class SqrLoop: public smoc_actor {
 2 public:
 3   smoc_port_in<double>  i1, i2;
 4   smoc_port_out<double> o1, o2;
 5 private:
 6   double v;
 7
 8   void copyStore()   {o1[0] = v = i1[0];}
 9   void copyInput()   {o1[0] = v;        }
10   void copyApprox()  {o2[0] = i2[0];    }
11   bool check() const {return fabs(v-i2[0]*i2[0])<0.01;}
12
13   smoc_firing_state start, loop;
14 public:
15   SqrLoop(sc_module_name name);
16 };
```

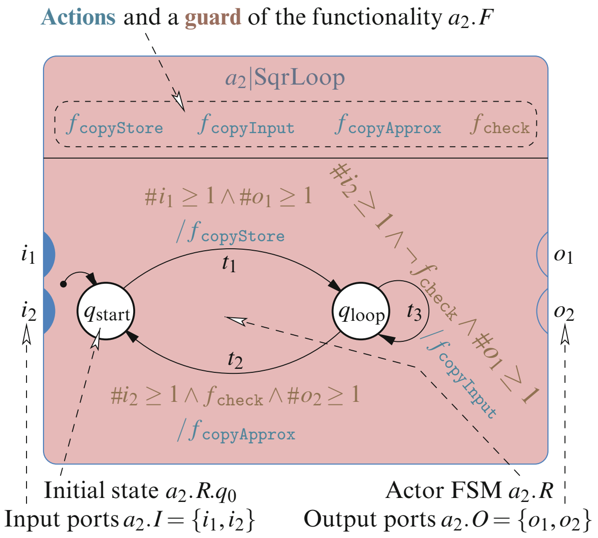

Actions and a guard of the functionality $a_2.F$
a_2|SqrLoop
$f_{\texttt{copyStore}}$ $f_{\texttt{copyInput}}$ $f_{\texttt{copyApprox}}$ $f_{\texttt{check}}$
$\#i_1 \geq 1 \wedge \#o_1 \geq 1$
$/f_{\texttt{copyStore}}$
t_1
$\#i_2 \geq 1 \wedge \neg f_{\texttt{check}} \wedge \#o_1 \geq 1$
$/f_{\texttt{copyInput}}$
i_1
i_2
o_1
o_2
q_{start}
q_{loop}
t_3
t_2
$\#i_2 \geq 1 \wedge f_{\texttt{check}} \wedge \#o_2 \geq 1$
$/f_{\texttt{copyApprox}}$
Initial state $a_2.R.q_0$
Actor FSM $a_2.R$
Input ports $a_2.I = \{i_1, i_2\}$
Output ports $a_2.O = \{o_1, o_2\}$

Fig. 3.8 The `SqrLoop` actor [10] from the network graph shown in Fig. 3.7 is composed of a set of *input ports* I and a set of *output ports* O, its *functionality* F, and its *FSM* R. The functionality can be further subdivided into actions (*colored cyan*) and guards (*colored brown*) as used by the FSM

3.3.3 Specification of the Communication Behavior

The *communication behavior* of an actor is separated strictly from the functional behavior of the actor on purpose. The communication behavior describes how and under what condition tokens are consumed and produced by an actor. In SysteMoC, the abstraction is done by representation of the communication behavior of an actor by its actor FSM, e.g., as seen in Fig. 3.8. More formally, a SysteMoC actor can be defined as follows:

Definition 2 (Actor [11]). An actor is a tuple $a = (I, O, F, R)$ containing a set of *actor input ports* I and *actor output ports* O, the *actor functionality* $F = F_{action} \cup F_{guard}$ partitioned into a set of actions and a set of guards, as well as the *actor FSM* R that is determining the *communication behavior* of the actor.

To exemplify, the methods `copyStore`, `copyInput`, and `copyApprox` (see Lines 8 to 10 in Listing 2) correspond to their respective actions, i.e., $f_{\text{copyStore}}$, $f_{\text{copyInput}}$, $f_{\text{copyApprox}} \in F_{action}$. Finally, the guard $f_{\text{check}} \in F_{guard}$ is represented by the `const` method `check` declared in Line 11.

For example, in state q_{start}, transition t_1 may be taken if there exists at least one token on input port i_1 ($\#i_1 \geq 1$) and at least one free place on the FIFO connected to output port o_1 ($\#o_1 \geq 1$). Once this condition is fulfilled, t_1 is taken and the action $f_{\text{copyStore}}$ executed. The transition or *actor firing* ends with the consumption of one token from i_1 and production of one token at the output port o_1 with the value computed according to the function $f_{\text{copyStore}}$ shown in Listing 2. The actor FSM

Listing 3 SysteMoC implementation of the SqrLoop actor a_2

```
 1 class SqrLoop: public smoc_actor {
 2   ...
 3   smoc_firing_state start, loop;
 4 public:
 5   SqrLoop(sc_module_name name)
 6     : smoc_actor(name, start),
 7       i1("i1"), i2("i2"), o1("o1"), o2("o2"),
 8       start("start"), loop("loop")
 9   {
10     start =
11         i1(1)                                 >>
12         o1(1)                                 >>
13         SMOC_CALL(SqrLoop::copyStore)         >> loop
14       ;
15     loop  =
16         (i2(1) &&  SMOC_GUARD(SqrLoop::check)) >>
17         o2(1)                                 >>
18         SMOC_CALL(SqrLoop::copyApprox)        >> start
19       |
20         (i2(1) && !SMOC_GUARD(SqrLoop::check)) >>
21         o1(1)                                 >>
22         SMOC_CALL(SqrLoop::copyInput)         >> loop
23       ;
24   }
25 };
```

of the `SqrLoop` actor as depicted in Fig. 3.8 is constructed by the code shown in Listing 3 Lines 10 to 23.

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

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

The transition t_1 from q_{start} to q_{loop} is given in Lines 11 to 13. Here, guards are again colored brown while actions are colored cyan. In detail, we use the syntax `i(n)` and `o(m)` to express the condition that at least n tokens, respectively, at least m free places must be present on the channel connected to the input port i and the channel connected to the output port o. Methods of the class used as actions or guard functions must be marked via usage of the `SMOC_CALL` or

`SMOC_GUARD` macros, respectively. Transitions t_2 and t_3 are defined accordingly in Lines 16 to 18 and Lines 20 to 22. As can be seen, SysteMoC—in contrast to KPN [24] and FunState (The model was initially published [42] as State Machine Controlled Flow Diagrams (SCF), but in later literature it is referred to as FunState, which is a short for Functions Driven by State Machines.) [41]—distinguishes its functions further into actions $f_{action} \in F_{action}$ and guards $f_{guard} \in F_{guard}$. To exemplify, the `SqrLoop` actor depicted in Fig. 3.8 is considered. This actor has three actions $F_{action} = \{ f_{\text{copyStore}}, f_{\text{copyInput}}, f_{\text{copyApprox}} \}$ and one guard function $F_{guard} = \{ f_{\text{check}} \}$. Naturally, the guard f_{check} is only used in a transition guard of the FSM R while the actions of the FSM are drawn from the set of actions F_{action}.

In SysteMoC, each FSM state is defined explicitly by an assignment of a list of outgoing transitions. To exemplify, the state q_{start} is defined in Listing 3 Line 10 by assigning transition t_1 (Lines 11 to 13) to it. If a state has multiple outgoing transitions, e.g., state q_{loop} with the transitions t_2 (Lines 16 to 18) and t_3 (Lines 20 to 22), then the outgoing transitions are joined via the "|"-operator (Line 19) and assigned to the state variable (Line 15).

3.4 Semantics and Execution Behavior of SysteMoC

More formally, an actor FSM resembles the FSM definition from FunState [41], but with slightly different definitions for actions and guards and defined as follows:

Definition 3 (Actor FSM [11]). The FSM R of an actor a is a tuple (Q, q_0, T) containing a finite set of *states* Q, an *initial state* $q_0 \in Q$, and a finite set of *transitions* T. A *transition* $t \in T$ itself is a tuple $(q_{\text{src}}, k, f_{action}, q_{\text{dst}})$ containing the source state $q_{\text{src}} \in Q$, from where the transition is originating, and the destination state $q_{\text{dst}} \in Q$, which will become the next current state after the execution of the transition starting from the current state q_{src}. Furthermore, if the transition t is taken, then an action f_{action} from the set of functions of the *actor functionality* $a.F_{action}$ will be executed. (We use the "."-operator, e.g., $a.F_{action}$, for member access of tuples whose members have been explicitly named in their definition, e.g., member F_{action} of the actor a from Definition 2.) Finally, the execution of the transition t itself is guarded by the *guard* k.

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

An actor is enabled if and only if it has at least one enabled transition. The firing of an actor corresponds to a non-deterministic selection and execution of one transition out of the set of enabled transitions of the actor. In general, if multiple actors have enabled transitions, then the transition, and hence its corresponding actor, is chosen non-deterministically out of the set of all enabled transitions in all actors. In summary, the execution of a SysteMoC model can be divided into three phases:

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

In contrast to FunState [41], a guard k is more structured. Moreover, it is partitioned into the following three components:

- The *input guard* which encodes a conjunction of input predicates on the number of available tokens on the input ports, e.g., $\#i_1 \geq 1$ denotes an input predicate that tests if at least one token is available at the actor input port i_1. Hence, consumption rates can be associated with each transition by the $\mathbf{cons} : (T \times I) \to \mathbb{N}_0$ function that determines for each transition and input port/channel combination the number of tokens that have to be at least present to enable the transition.
- The *output guard* which encodes a conjunction of output predicates on the number of free places on the output ports, e.g., $\#o_1 \geq 1$ denotes an output predicate that tests if at least one free place is available at the actor output port o_1. Thus, production rates can be associated with each transition by the $\mathbf{prod} : (T \times O) \to \mathbb{N}_0$ function that specifies for each transition and output port/channel combination the number of free places that must be at least available to enable the transition.
- The *functionality guard* which encodes a logical composition of guard functions of the actor, e.g., $\neg f_{\texttt{check}}$ annotated to the transition t_3 of the actor FSM of the actor a_2 in Fig. 3.8. Hence, the *functionality guard* depends on the *functionality state* and the token values on the input ports.

Here, the notion of a *functionality state* $q_{\text{func}} \in Q_{\text{func}}$ of an actor is used. The functionality state is an abstract representation of the C++ variables in the real implementation of a SysteMoC actor. This functionality state is required for notational completeness, that is for the formal definitions of action and guard functions, i.e., $f_{action} : Q_{\text{func}} \times S^{|I|} \to Q_{\text{func}} \times S^{|O|}$ and $f_{guard} : Q_{\text{func}} \times S^{|I|} \to \{\mathbf{t}, \mathbf{f}\}$. (The usual notation $|X|$ is used to denote the cardinality of a set X.) Both types of functions depend on the functionality state of their actor. Furthermore, an action may update this state, while a guard function may not. Hence, all SysteMoC actor firings are sequentialized over this functionality state. Therefore, multiple actor firings of the same actor cannot be executed in parallel. That is, in data-flow parlance, all SysteMoC actors have a virtual self-loop with one initial token that corresponds to $q_{\text{func}} \in Q_{\text{func}}$. However, as the functionality state is not used for any optimization and analysis algorithms presented in this chapter, it has been excluded from Definition 2.

Note that the consumption and production rate functions also specify the number of tokens that are consumed/produced on the input/output ports if a transition t is taken. That is, $\forall i \in I : \mathbf{cons}(i, t.f_{action}) = \mathbf{cons}(t, i) \wedge \forall f_{guard}$ contained in $t.k$: $\mathbf{cons}(i, f_{guard}) \leq \mathbf{cons}(t, i)$ and $\forall o \in O : \mathbf{prod}(t.f_{action}, o) = \mathbf{prod}(t, o)$. Hence, for a SysteMoC model to be *well formed*, the number of tokens accessed on the different input and output ports by an action $t.f_{action}$ associated with the transition t as well as the guard functions f_{guard} used in the transition guard $t.k$ has to conform to the consumption and production rates of the transition.

We enforce this requirement by checking that access to tokens by actions and guards via the ports, e.g., the syntax `i[n]` and `o[m]` that is used in Fig. 3.9, Listing 2 to access the `n`th token relative to the read pointer, respectively, the `m`th token relative to the write pointer, does not access tokens outside the so-called *random-access region* that is controlled by the actor FSM as will be explained in the following. Hence, as only tokens inside the random-access region can be modified by the actions or read by the guards, the actor FSM fully controls the communication behavior of the actor.

To exemplify, we consider a simple example with just one source and one sink actor connected by a single channel as depicted in Fig. 3.9. Here, the random-access regions are marked by bold brown-bordered boxes.

Both actors a_1 and a_2 are in the process of executing their respective actions f_{sink} and f_{src}. Thus, the guards $\#i_1 \geq 3$ and $\#o_1 \geq 2$ are satisfied, but the three tokens and two free places are not yet consumed and produced, respectively, but only reserved for consumption and production when execution of the actions finishes. Then, the read and write pointers of the FIFO channels will also be advanced by the actor FSMs by three, respective, two, tokens. Before the execution of the actions is started, the random-access regions are sized according to the guards of the transitions that are currently taken. During the execution of the actions, random access and update of all data values of the tokens contained in the random-access regions—but not outside these regions—is allowed by the executing actions.

Finally, for a SysteMoC model to be well formed, no sharing of state between actors via global variables is allowed. This requirement cannot be enforced by the SysteMoC library itself and is also usually not a problem in the code of the actor itself, but in library code that is used by the actor. A discussion of modeling library dependencies via *library tasks* and the problem of persistent states inside these libraries is tackled by the Common Intermediate Code (CIC) model that is introduced in ▶ Chap. 29, "HOPES: Programming Platform Approach for Embedded Systems Design".

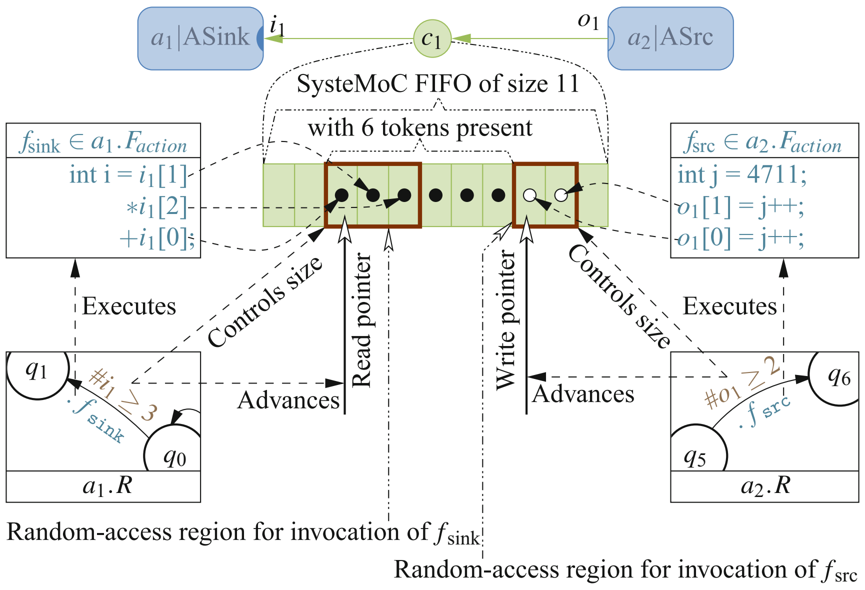


Fig. 3.9 Depicted above is a simple source-sink example that is used to explain the semantics of a SysteMoC FIFO associated with the channel c_1. The buffer memory of the channel c_1 is organized as a *ring buffer* with its associated *read and write pointers*. The buffer memory has a capacity of 11 tokens (depicted as *light green cells* inside the *green-bordered box* representing the buffer memory) and is already filled with 6 tokens (the 6 *solid black dots*). From the remaining five free places, two places have been reserved (the two *black-bordered* but *white-filled dots*) by the actor FSM $a_2.R$ for the execution of the action f_{src}. These reserved places correspond to tokens that have not yet been associated with data values ν. It is the responsibility of the action to compute the missing data values. Finally, for each function invocation and for each channel accessed by the invoked function, a random-access region is defined for accessing the channel. Hence, two random-access regions (depicted as *bold brown-bordered boxes*) are shown in the above figure

3.5 Analysis of SysteMoC Models

SysteMoC permits modeling of non-determinate data-flow graphs. However, when using SysteMoC models as an input to Hardware/Software Codesign (HSCD) flows, it is advantageous for reason of analyzability to identify parts of the model, which belong to restricted MoCs. Thanks to its formal representation of the firing behavior of an actor by an actor FSM, MoC identification can be performed within SysteMoC for its actors. In the following, we present the representation as well as identification of SDF and CSDF actors. The ability of classification [12,45] serves as a distinction between SysteMoC and Ptolemy [8,23,35] as well as ForSyDe, which is introduced in ▶ Chap. 4, "ForSyDe: System Design Using a Functional Language and Models of Computation", and similar approaches that use distinct modeling libraries for each different MoC. For example, in Ptolemy each actor is associated with a director which explicitly specifies the MoC under which the actor is executed. This is important as the data-flow model of SysteMoC is chosen in order to provide greatest expressiveness. Hence, the analyzability of a general SysteMoC specification (e.g., with respect to deadlock freedom or the ability to be executed in bounded memory) is limited. Nonetheless, only very simple actors will be classified as belonging to the static data-flow domain. Hence, a more detailed discussion as well as identification of BDF and DDF actors can be found in [44]. Scheduling algorithms that take advantage of this information are provided in [5] for the BDF MoC. Optimizations for more expressive MoCs is an ongoing topic of current research.

3.5.1 Representing SDF and CSDF Actors in SysteMoC

As SysteMoC permits modeling of quite general types of data-flow graphs, it is also possible to represent any SDF and CSDF actor in SysteMoC. Given a CSDF graph as defined in Definition 1 and using the ancillary definitions of the **cons** and **prod** functions as described in Sect. 3.2.2.3, then for a given CSDF actor a, a functionally equivalent SysteMoC actor a' can be constructed as follows: First, the input ports I and output ports O of the SysteMoC actor a' correspond to the incoming and outgoing channels of the CSDF actor a, respectively. Assuming that this actor has τ CSDF phases, then the corresponding actor FSM R is built by generating a state for each phase of the CSDF actor, i.e., $Q = \{q_0, q_1, \ldots q_{\tau-1}\}$, and marking the state q_0 as the initial state of the FSM. Next, these states are connected such that the resulting transitions form a single cycle, i.e., $T = \{(q_0, k_0, f_{\texttt{phase}_0}, q_1), (q_1, k_1, f_{\texttt{phase}_1}, q_2), \ldots (q_{\tau-1}, k_{\tau-1}, f_{\texttt{phase}_{\tau-1}}, q_0)\}$. As an example, the resulting FSM R of the CSDF actor a_3 is shown in Fig. 3.10b. Here, the actions $f_{\texttt{phase}_0}, f_{\texttt{phase}_1}, \ldots f_{\texttt{phase}_{\tau-1}}$ are associated with the firing of the CSDF actor in the corresponding phase. The token consumption and production rates encoded by the guards $k_0, k_1, \ldots k_{\tau-1}$ and given by the functions **cons** : $(T \times I) \rightarrow \mathbb{N}_0$ and **prod** : $(T \times O) \rightarrow \mathbb{N}_0$ of the SysteMoC model can be directly derived from the functions **cons** : $C \rightarrow \mathbb{N}_0^\tau$ and **prod** : $C \rightarrow \mathbb{N}_0^\tau$ of the CSDF graph. Remember that we use the notion of ports interchangeably with the channels connected to these ports. Hence, we can define the **cons** and **prod** functions of the SysteMoC model as follows:

$$\mathbf{prod}(t_l, o) = n_l \quad \text{where } (n_0, n_1, \ldots n_{\tau-1}) = \mathbf{prod}(o) \quad \forall l \in \{0, 1, \ldots \tau - 1\}, o \in O$$
$$\mathbf{cons}(t_l, i) = m_l \quad \text{where } (m_0, m_1, \ldots m_{\tau-1}) = \mathbf{cons}(i) \quad \forall l \in \{0, 1, \ldots \tau - 1\}, i \in I$$

To exemplify, for the CSDF actor a_3 from Fig. 3.10c, the resulting normalized actor FSM is shown in Fig. 3.10b. Corresponding to the first phase, transition t_0 consumes one token from i_1 and produces seven tokens on o_1. The second phase embodied by transition t_1 consumes two tokens from i_1 and produces one token on o_1. The remaining phases are derived accordingly.

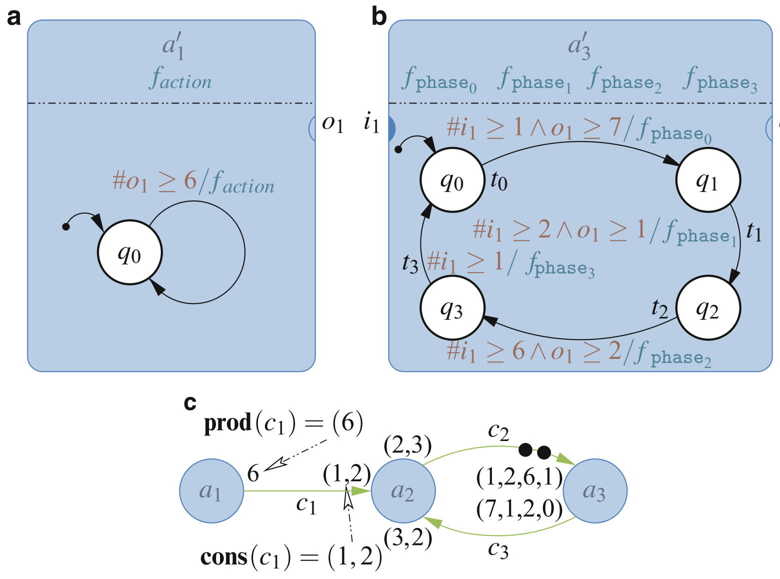

a
a'_1
f_{action}
o_1
$\#o_1 \geq 6/f_{action}$
q_0
b
a'_3
f_{phase_0} f_{phase_1} f_{phase_2} f_{phase_3}
i_1
o_1
$\#i_1 \geq 1 \wedge o_1 \geq 7/f_{\text{phase}_0}$
q_0
t_0
q_1
$\#i_1 \geq 2 \wedge o_1 \geq 1/f_{\text{phase}_1}$
t_1
t_3
$\#i_1 \geq 1/f_{\text{phase}_3}$
q_3
t_2
q_2
$\#i_1 \geq 6 \wedge o_1 \geq 2/f_{\text{phase}_2}$
c
$\textbf{prod}(c_1) = (6)$
6
(2,3)
c_2
(1,2)
a_1
a_2
(1,2,6,1)
(7,1,2,0)
a_3
c_1
(3,2)
c_3
$\textbf{cons}(c_1) = (1,2)$

Fig. 3.10 Example translations of an SDF and a CSDF actor into SysteMoC. (**a**) SysteMoC realization of the SDF actor a_1 from the DFG below. (**b**) SysteMoC realization of the CSDF actor a_3 from the DFG below. (**c**) Example of a CSDF graph also containing an SDF actor

3.5.2 SDF and CSDF Semantics Identification for SysteMoC Actors

In order to be able to apply MoC-specific analysis methods such as deadlock analysis or generation of static schedules, it is important to recognize individual actors or subgraphs thereof belonging to well-known data-flow models of computation such as HSDF, SDF, and CSDF. While it is possible to transform any static data-flow actor into a SysteMoC actor as described above, it is much more difficult to check if a given SysteMoC actor shows SDF or CSDF semantics. It will be shown that this can be accomplished by inspection and analysis of the *actor FSM* of a SysteMoC actor.

The most basic representation of static actors encoded as actor FSMs can be seen in Fig. 3.10. To exemplify, the SDF actor depicted in Fig. 3.10a contains only one transition, which produces six tokens onto the actor output port o_1. Clearly, the actor exhibits a static communication behavior corresponding to the SDF MoC. However, for the classification algorithm to be able to ignore the guard functions used by an actor FSM, certain assumptions have to be made. It will be assumed that given sufficient tokens and free places on the actor input and output ports, at least one of the guards of the outgoing transitions will evaluate to true, and, hence, there exists at least one enabled outgoing transition. This is required in order to be able to activate the equivalent SDF or CSDF actor an infinite number of times. It will also be assumed that an actor will consume or produce tokens every now and then. These assumptions are not erroneous as, otherwise, there exists an infinite loop in the execution of the actor where a cycle of the FSM is traversed and each transition in this cycle neither consumes nor produces any tokens. This can clearly be identified as a bug in the implementation of the actor, similar to an action of a transition that never terminates. For the exact mathematical definition of the above-given requirements, see [45].

The idea of the classification algorithm is to check if the communication behavior of a given actor can be reduced to a basic representation, which can be easily classified into the SDF or CSDF MoC. Note that the basic representations for both SDF and CSDF models are FSMs with a single cycle, e.g., as depicted for the CSDF actor in Fig. 3.10b.

Yet, not all *actor FSMs* satisfy this condition (see Fig. 3.11). However, some of them, e.g., Fig. 3.11b, c, still show the communication behavior of a static actor. It can be distinguished if the analysis of the *actor functionality state* (see Definition 2) is required to decide whether an actor is a static actor, e.g., Fig. 3.11c, or not, e.g., Fig. 3.11b. In the following, the actor functionality state will not be considered. Therefore, the presented classification algorithm will fail to classify Fig. 3.11c as a static actor, but will achieve the classification as static for the actor shown in Fig. 3.11b. In that sense, the algorithm only provides a *sufficient* criterion for the problem of static actor detection. A *sufficient* and *necessary* criterion cannot be given as the problem is undecidable in the general case.

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

Definition 4 (Classification Candidate [45]). A possible CSDF behavior of a given actor is captured by a *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1, \ldots \mu_{\tau-1} \rangle$ where each $\mu = (\text{cons}, \text{prod})$ represents a phase of the CSDF behavior and τ is the number of phases.

To exemplify, Fig. 3.10b is considered. In this case, four phases are present, i.e., $\tau = 4$, where the first phase (μ_0) consumes one token from i_1 and produces seven tokens on o_1, the second phase (μ_1) consumes two tokens from i_1 and produces one token on o_1, the third phase (μ_2) consumes six tokens and produces two, and the final phase (μ_3) only consumes one token.

It can be observed that all paths starting from the initial *actor FSM* state q_0 must comply with the *classification candidate*. Furthermore, as CSDF exhibits cyclic behavior, the paths must also contain a cycle. The classification algorithm will search for such a path $p = \langle t_1, t_2, \ldots, t_n \rangle$ of transitions t_i in the *actor FSM* starting from the initial state q_0. The path can be decomposed into an acyclic prefix path p_a and a cyclic path p_c such that $p = p_a^\frown p_c$, i.e., $p_a = \langle t_0, t_1, \ldots t_{l-1} \rangle$ being the

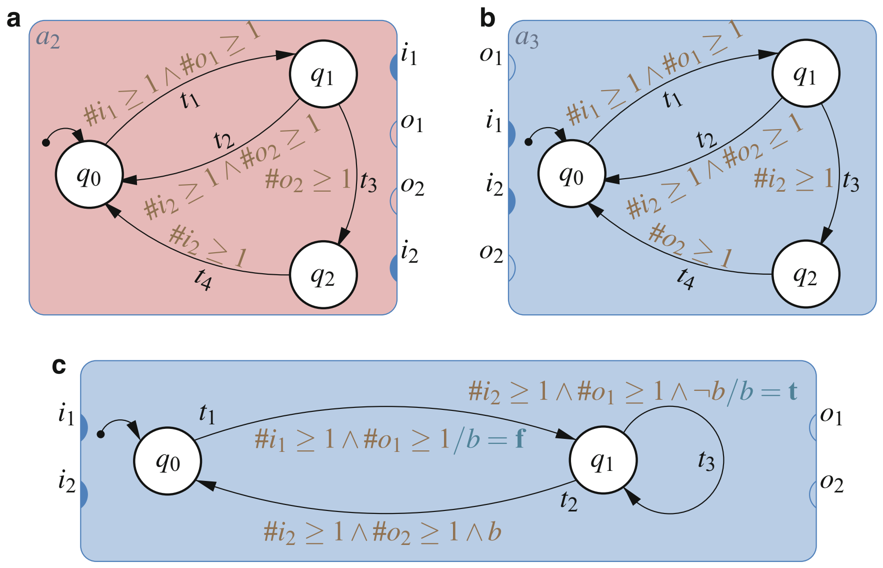

a
a_2
i_1
o_1
o_2
i_2
q_0
q_1
q_2
t_1
$\#i_1 \geq 1 \wedge \#o_1 \geq 1$
t_2
$\#i_2 \geq 1 \wedge \#o_2 \geq 1$
$\#o_2 \geq 1$
t_3
$\#i_2 \geq 1$
t_4
b
a_3
o_1
i_1
i_2
o_2
$\#i_2 \geq 1$
$\#o_2 \geq 1$
c
$\#i_2 \geq 1 \wedge \#o_1 \geq 1 \wedge \neg b / b = \mathbf{t}$
$\#i_1 \geq 1 \wedge \#o_1 \geq 1 / b = \mathbf{f}$
$\#i_2 \geq 1 \wedge \#o_2 \geq 1 \wedge b$

Fig. 3.11 Various actor FSMs that do not match the basic representation of static actors as exemplified in Fig. 3.10. (**a**) Visualized above is the actor a_2 with an *actor FSM* that cannot be converted to a basic representation of static actors. (**b**) Shown is the actor a_3 containing an *actor FSM* that can be converted to basic CSDF representation. (**c**) Shown is an actor that seems to belong to the *dynamic domain*, but exhibits CSDF behavior due to the manipulation of the Boolean variable b. Hence, leading to a cyclic execution of the transitions t_1, t_3, and t_2 of its *actor FSM*

prefix and $p_c = \langle t_l, t_{l+1}, \dots t_{n-1} \rangle$ being the cyclic part, that is $t_l.q_{\text{src}} = t_{n-1}.q_{\text{dst}}$. After such a path p has been found, a set of *classification candidates* can be derived from the set of all nonempty prefixes $\{p' \sqsubseteq p \mid \#p' \in \{1, 2, \dots n\}\}$ of the path p. (Here, the "$\sqsubseteq$"-operator denotes that a sequence is a prefix of some other sequence, e.g., $\langle 3, 1 \rangle \sqsubseteq \langle 3, 1 \rangle \sqsubseteq \langle 3, 1, 2 \rangle \sqsubseteq \langle 3, 1, 2, \dots \rangle$, but $\langle 3, 1 \rangle \not\sqsubseteq \langle 4, 1, 2 \rangle$ as well as $\langle 3, 1 \rangle \not\sqsubseteq \langle 3 \rangle$.) A *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1, \dots \mu_{\tau-1} \rangle$ is derived from a nonempty prefix p' by (1) unifying adjacent transitions t_j, t_{j+1} according to the below given *transition contraction condition* until no more contractions can be performed and (2) deriving the CSDF phases μ_j of the *classification candidate* from the transitions t_j of the contracted path computed in step (1).

Definition 5 (Transition Contraction Condition [12]). Two transitions t_j and t_{j+1} of a prefix path p' can be contracted if t_j only consumes tokens, i.e., $\mathbf{prod}(t_j, O) = \mathbf{0}$, or t_{j+1} only produces tokens, i.e., $\mathbf{cons}(t_{j+1}, I) = \mathbf{0}$. (For notational brevity, the construct $\mathbf{cons}(t, I) = (n_{i_1}, n_{i_2} \dots, n_{i_{|I|}}) = \boldsymbol{n}_{\text{cons}}$ is used to denote the vector $\boldsymbol{n}_{\text{cons}}$ of numbers of tokens consumed by taking the transition. The equivalent notation $\mathbf{prod}(t, O)$ is also used similarly for produced tokens.) The resulting transition t' has the combined consumption and production rates given as $\mathbf{cons}(t', I) = \mathbf{cons}(t_j, I) + \mathbf{cons}(t_{j+1}, I)$ and $\mathbf{prod}(t', O) = \mathbf{prod}(t_j, O) + \mathbf{prod}(t_{j+1}, O)$.

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

```
1:  function VALIDATECLASSCAND($\boldsymbol{\mu}$, $R$)
2:      $\tau = \#\boldsymbol{\mu}$                                        ▷ Number of CSDF phases $\tau$
3:      $\langle \mu_0, \mu_1, \dots \mu_{\tau-1} \rangle = \boldsymbol{\mu}$                ▷ CSDF phases $\mu_0, \mu_1, \dots \mu_{\tau-1}$
4:      $\omega_0 = (0, \mu_0.\text{cons}, \mu_0.\text{prod})$                    ▷ Initial annotation tuple
5:      queue ← $\langle\rangle$                ▷ Set up the empty queue for breadth-first search
6:      ann ← $\emptyset$                          ▷ Set up the empty set of annotations
7:      queue ← queue$^\frown(R.q_0)$                                  ▷ Enqueue $R.q_0$
8:      ann ← ann$\cup\{(R.q_0, \omega_0)\}$                         ▷ Annotate the initial tuple
9:      while #queue > 0 do                              ▷ While the queue is not empty
10:         $q_{\text{src}}$ = hd(queue)                                  ▷ Get head of queue
11:         $\omega_{\text{src}}$ = ann($q_{\text{src}}$)                  ▷ Get annotation tuple for state $q_{\text{src}}$
12:         queue ← tl(queue)                            ▷ Dequeue head from queue
13:         for all $t \in R.T$ where $t.q_{\text{src}} = q_{\text{src}}$ do
14:             if $\omega_{\text{src}}.\text{cons} \not\geq \mathbf{cons}(t, I) \vee \omega_{\text{src}}.\text{prod} \not\geq \mathbf{prod}(t, O)$ then
15:                 return f                ▷ Reject $\boldsymbol{\mu}$ due to failed transition criterion I
16:             end if
17:             if $\omega_{\text{src}}.\text{cons} \neq \mathbf{cons}(t, I) \wedge \mathbf{prod}(t, O) \neq \mathbf{0}$ then
18:                 return f               ▷ Reject $\boldsymbol{\mu}$ due to failed transition criterion II
19:             end if
20:             $\omega_{\text{dst}}$ ← derive from $\omega_{\text{src}}$ and $t$ as given by Equation (3.8)
21:             if $\exists \omega'_{\text{dst}} : (t.q_{\text{dst}}, \omega'_{\text{dst}}) \in$ ann then          ▷ Annotated tuple present?
22:                 if $\omega'_{\text{dst}} \neq \omega_{\text{dst}}$ then             ▷ Check annotated tuple for consistency
23:                     return f                     ▷ Reject classification candidate $\boldsymbol{\mu}$
24:                 end if
25:             else                                ▷ No tuple annotate to state $t.q_{\text{dst}}$
26:                 ann ← ann$\cup\{(t.q_{\text{dst}}, \omega_{\text{dst}})\}$                   ▷ Annotate tuple $\omega_{\text{dst}}$
27:                 queue ← queue$^\frown(t.q_{\text{dst}})$                           ▷ Enqueue $t.q_{\text{dst}}$
28:             end if
29:         end for
30:     end while
31:     return t                                    ▷ Accept classification candidate $\boldsymbol{\mu}$
32: end function
```

For clarification, Fig. 3.11a, b are considered and it is assumed that the path $p = \langle t_1, t_3, t_4 \rangle$ has been found. Hence, the set of all nonempty prefixes is $\{ \langle t_1 \rangle, \langle t_1, t_3 \rangle, \langle t_1, t_3, t_4 \rangle \}$. In both depicted FSMs, the transition t_2 consumes and produces exactly as many tokens as the transition sequence $\langle t_3, t_4 \rangle$. However, the transition sequence $\langle t_3, t_4 \rangle$ in Fig. 3.11b can be contracted as t_3 only consumes tokens while transition sequence $\langle t_3, t_4 \rangle$ in Fig. 3.11a cannot be contracted. Hence, for Fig. 3.11b, the *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1 \rangle$ derived from the prefix $p' = \langle t_1, t_3, t_4 \rangle$ is as follows:

$$\begin{aligned}
\mu_0.\text{cons} &= \mathbf{cons}(t_1, I) &&= (1, 0) \\
\mu_0.\text{prod} &= \mathbf{prod}(t_1, O) &&= (1, 0) \\
\mu_1.\text{cons} &= \mathbf{cons}(t_3, I) + \mathbf{cons}(t_4, I) &&= (0, 1) \\
\mu_1.\text{prod} &= \mathbf{prod}(t_3, O) + \mathbf{prod}(t_4, O) &&= (0, 1)
\end{aligned}$$

On the other hand, for Fig. 3.11a, the *classification candidate* $\boldsymbol{\mu} = \langle \mu_0, \mu_1, \mu_2 \rangle$ derived from the prefix $p' = \langle t_1, t_3, t_4 \rangle$ does not exhibit any contractions and is depicted below as:

$$\begin{array}{ll} \mu_0.\text{cons} = (1,0) & \mu_0.\text{prod} = (1,0) \\ \mu_1.\text{cons} = (0,0) & \mu_1.\text{prod} = (0,1) \\ \mu_2.\text{cons} = (0,1) & \mu_2.\text{prod} = (0,0) \end{array}$$

The underlying reasons for the *transition contraction condition* from Definition 5 are discussed in the following. To illustrate, the data-flow graph depicted in Fig. 3.12a which uses two actors a_2 and a_3 containing the FSMs from Fig. 3.11a, b is considered. Obviously, there exist dependencies between the transitions in a legal transition trace of the actors a_2 and a_3 as shown in Fig. 3.12b.

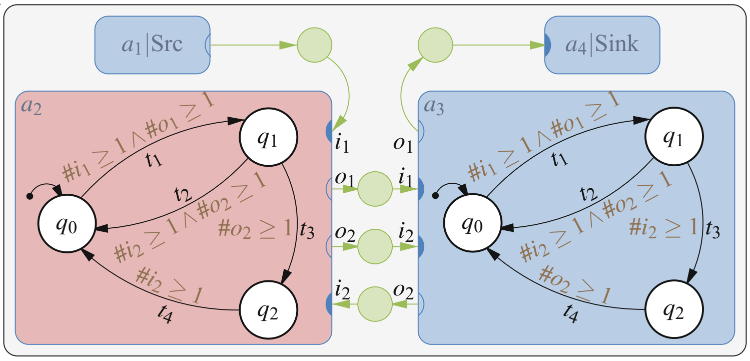
For example, it can be seen in Fig. 3.12c that if the transition sequence $\langle t_3, t_4 \rangle$ of actor a_3 is contracted into a single transition t_c, then the resulting dependencies of t_c are exactly the same as for transition t_2. This is the reason why the FSM from Fig. 3.11b can be classified into a CSDF actor. Furthermore, the contraction is a valid transformation as it does not change the set of possible transition sequences of the whole data-flow graph, apart from the substitution of the transition sequence $\langle t_3, t_4 \rangle$ by the transition t_c. This can be seen by comparing the possible transition sequences depicted in Fig. 3.12b, c. Compacting the transition sequence $\langle t_3, t_4 \rangle$ of actor a_3 generates the transition t_c, which is inducing the data dependency that the token produced on port o_2 by the transition t_c depends on the token consumed on port i_2 by the transition t_c. However, the previous transition sequence $\langle t_3, t_4 \rangle$ also induces this data dependency as t_4 can only be taken after t_3.

In contrast to this, the contraction of the transition sequence $\langle t_3, t_4 \rangle$ of actor a_2 into a transition t_d does introduce a new erroneous data dependency from the consumption of a token on i_2 to the production of a token on o_2. The data dependency is erroneous as the original transition sequence $\langle t_3, t_4 \rangle$ has no such dependency as it first produces the token on o_2 before trying to consume a token on i_2. Indeed, an erroneous contraction might introduce a deadlock into the system as can be seen in Fig. 3.12d where the transition t_d is part of two dependency cycles $a_2.t_d \rightarrow a_3.t_2 \rightarrow a_2.t_d$ and $a_2.t_d \rightarrow a_3.t_3 \rightarrow a_3.t_4 \rightarrow a_2.t_d$ which are not present in the original dependency structure depicted in Fig. 3.12b.

After the set of *classification candidates* has been derived from the actor FSM, each candidate is checked for validity by Algorithm 1. The checks are performed sequentially starting from the *classification candidate* derived from the shortest prefix p' to the candidate derived from the full path p. If a candidate is accepted by Algorithm 1, then the actor is a CSDF actor with phases as given by the accepted *classification candidate* $\boldsymbol{\mu}$.

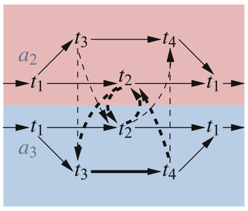
The main idea of the validation algorithm is to check a *classification candidate* against all possible transition sequences reachable from the initial state of the actor FSM. However, due to the existence of contracted transitions in the *classification candidate* as well as in the actor FSM, a simple matching of a phase μ_j to

a



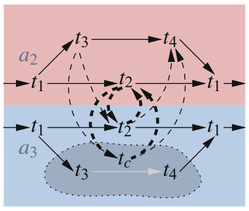
Shown is a DFG containing the actors a_2 and a_3 from Figure 3.11. Due to mutual dependencies between the actor firings of the actors a_2 and a_3, actor a_2 will never execute transition t_2. On the other hand, actor a_3 is free to choose either the transition sequence $\langle t_3, t_4 \rangle$ or the transition t_2 in its execution trace.

b



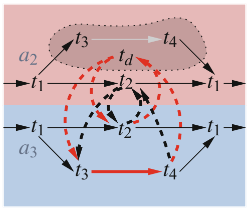
Shown are the dependencies in the transition trace of actor a_2 and a_3. Two dependency cycles are already present (marked bold black) that cause the inability to fire transition t_2 of actor a_2.

c



Shown are the dependencies for the contraction of the transition sequence $\langle t_3, t_4 \rangle$ of actor a_3 into the transition t_c. One can observe that no additional de-pendency cycles have been introduced by this contraction.

d



Shown are the dependencies for the contraction of the transition sequence $\langle t_3, t_4 \rangle$ of actor a_2 into the transition t_d. Thus, inducing two (marked bold red) erroneous dependency cycles.

Fig. 3.12 Given (see (**a**)) is an example DFG containing two actors a_2 and a_3 used to show valid and invalid transition contractions. For this purpose, dependencies in transition sequences of both actors are shown in (**b**), (**c**), and (**d**). Here, *dashed lines* represent dependencies induced by the availability of data (tokens) and *solid lines* represent dependencies induced by the sequential nature of the FSM. Dependency cycles in these transition traces are marked by bolding the corresponding edges

an FSM transition is infeasible. Instead, a CSDF phase μ_j is matched by a transition sequence. To keep track of the number of tokens already produced and consumed for a CSDF phase, each FSM state q_n will be annotated with a tuple $\omega_n = (j, \text{cons}, \text{prod})$ where cons and prod are the vectors of number of tokens which are still to be consumed and produced to complete the consumption and production of the CSDF phase μ_j and j denotes the CSDF phase which should be matched.

The validation algorithm starts by deriving the tuple ω_0 (see Line 4) for the initial state q_0 from the first CSDF phase μ_0 of the *classification candidate* $\boldsymbol{\mu}$. The algorithm uses the set ann as a function $\omega_n = \texttt{ann}(q_n)$ from an FSM state to its annotated tuple ω_n. Initially, the $\texttt{ann}$ set is empty (see Line 6) denoting that all function values $\texttt{ann}(q_n)$ are undefined, and hence no tuples have been annotated. The annotation of tuples starts with the initial tuple $\omega_0 = \texttt{ann}(q_0)$ for the initial state q_0 by adding the corresponding association to the $\texttt{ann}$ set in Line 8.

The algorithm proceeds by performing a *breadth-first search* (see Lines 5, 9, 12 and 27) of all states q_n of the FSM R starting from the initial state q_0 (see Line 7). For each visited state q_{src} (see Line 10) its annotated tuple ω_{src} will be derived from the set $\texttt{ann}$ (see Line 11) and the corresponding outgoing transitions t of the state q_{src} (see Line 13) are checked against the annotated tuple ω_{src} (see Lines 14 to 19) via the transition criteria as given below:

Definition 6 (Transition Criterion I [45]). Each outgoing transition $t \in T_{\text{src}} = \{t \in T \mid t.q_{\text{src}} = q_{\text{src}}\}$ of a visited FSM state $q_{\text{src}} \in Q$ must consume and produce less or equal tokens than specified by the annotated tuple ω_{src}, i.e., $\forall t \in T_{\text{src}} : \omega_{\text{src}}.\text{cons} \geq \mathbf{cons}(t, I) \wedge \omega_{\text{src}}.\text{prod} \geq \mathbf{prod}(t, O)$. Otherwise, the annotated tuple ω_{src} is invalid and the *classification candidate* $\boldsymbol{\mu}$ will be rejected.

Definition 7 (Transition Criterion II [45]). Each outgoing transition $t \in T_{\text{src}} = \{t \in T \mid t.q_{\text{src}} = q_{\text{src}}\}$ of a visited FSM state $q_{\text{src}} \in Q$ must not produce tokens if there are still tokens to be consumed in that phase after the transition t has been taken, i.e., $\forall t \in T_{\text{src}} : \omega_{\text{src}}.\text{cons} = \mathbf{cons}(t, I) \vee \mathbf{prod}(t, O) = \mathbf{0}$. Otherwise, the annotated tuple ω_{src} is invalid and the *classification candidate* $\boldsymbol{\mu}$ will be rejected.

Transition criterion I ensures that a matched transition sequence consumes and produces exactly the number of tokens as specified by the CSDF phase μ of the *classification candidate*. *Transition criterion II* ensures that a transition sequence induces the same data dependencies as the CSDF phase μ of the *classification candidate*. If the transition does not conform to the above transition criteria, then the *classification candidate* is invalid and will be rejected (see Lines 15 and 18). Otherwise, that is conforming to both criteria, the matched transition sequence can be condensed via Definition 5 to the matched CSDF phase.

After the transition has been checked, the tuple ω_{dst} for annotation at the transition destination state $t.q_{\text{dst}}$ is computed in Line 20 according to the equation given below:

$$\begin{aligned} \text{cons}_{\text{left}} &= \omega_{\text{src}}.\text{cons} - \mathbf{cons}(t, I) \\ \text{prod}_{\text{left}} &= \omega_{\text{src}}.\text{prod} - \mathbf{prod}(t, O) \\ j' &= (\omega_{\text{src}}.j + 1) \bmod \tau \\ \omega_{\text{dst}} &= \begin{cases} (\omega_{\text{src}}.j, \text{cons}_{\text{left}}, \text{prod}_{\text{left}}) \text{ if } \text{cons}_{\text{left}} \neq \mathbf{0} \wedge \text{prod}_{\text{left}} \neq \mathbf{0} \\ (j', \mu_{j'}.\text{cons}, \mu_{j'}.\text{prod}) \text{ otherwise} \end{cases} \end{aligned} \tag{3.8}$$

In the above equation, $\text{cons}_{\text{left}}$ and $\text{prod}_{\text{left}}$ denote the consumption and production vectors remaining to match the CSDF phase μ_j. If these remaining consumption and production vectors are both the zero vector, then the matching of the CSDF phase μ_j

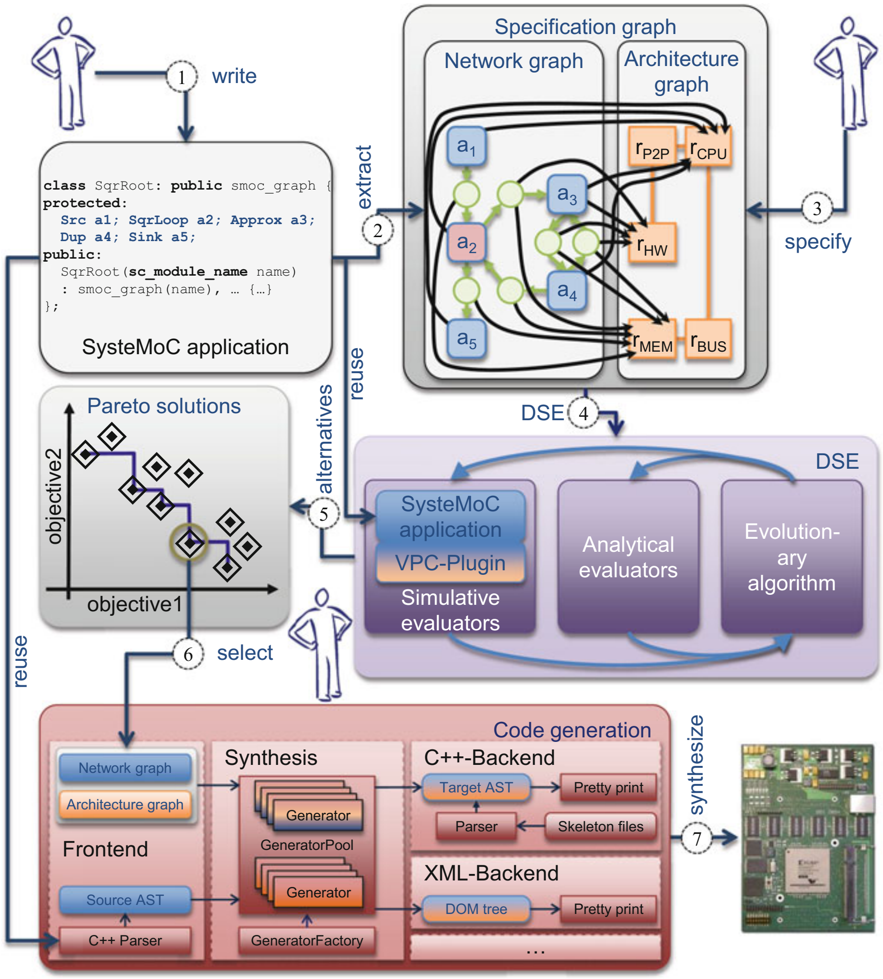
to a transition sequence has been completed. Hence, the tuple ω_{dst} will be computed to match the next CSDF phase $\mu_{(j+1) \bmod \tau}$. (The notation $n = l \bmod m$ for values $l \in \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}, m \in \mathbb{N} = \{1, 2, \ldots\}$ is used to denote the common residue $n \in \{0, 1, 2, \ldots m-1\}$ which is a non-negative integer smaller than m such that $l \equiv n \pmod{m}$.) Otherwise, the tuple ω_{dst} will use the updated remaining consumption and production vectors as given by $\mathrm{cons}_{\mathrm{left}}$ and $\mathrm{prod}_{\mathrm{left}}$.

Finally, if no tuple is annotated to the destination state $t.q_{\mathrm{dst}}$, then the computed tuple ω_{dst} is annotated to the destination state in Line 26 and the destination state is appended to the queue for the *breadth-first search* in Line 27. Otherwise, an annotated tuple is already present for the destination state (see Line 21). If this annotated tuple ω'_{dst} and the computed tuple ω_{dst} are inconsistent, then the classification candidate will be rejected (see Lines 21 to 23).

3.6 Hardware/Software Codesign with SysteMoC

SysteMoC is the core language of the SystemCoDesigner [15, 25] codesign framework and HSCD flow, which is briefly outlined in Fig. 3.13. In Step 1, the application to be implemented is written in SysteMoC. During an extraction step (Step 2 in Fig. 3.13), the SysteMoC actors and channels as well as their connections are automatically extracted from the SysteMoC application. As a result, the *network graph* as defined in Sect. 3.3 can be used as input to the DSE of the SystemCoDesigner flow. As a second input to DSE, the possible variety of architectures is modeled by an *architecture graph* [4] that is specified by the designer in Step 3. The architecture graph is composed of *resources* (shaded orange) and edges connecting two resources that are able to communicate with each other. Within SystemCoDesigner, resources are usually modeled at a granularity of processors, hardware accelerators, communication buses, centralized switches (crossbars), decentralized switches (Networks on Chip), memories. In this context, edges of the architecture graph are interpreted as links. Finally, *mapping edges* (black) are specified within Step 3 by the designer for each actor (bordered blue) and each channel (shaded green) of the network graph. A mapping edge indicates the option to implement and execute an actor on the resource it points to. For a channel, the corresponding mapping edge represents the possibility to use the associated resource as buffer for messages sent over the channel. Both graphs, i.e., network graph and architecture graph, together with the mapping edges form a *specification graph* [4] that serves as input for DSE in Step 4.

To exemplify the concept of a specification graph, we use again the network graph implementing Newton's iterative square root algorithm introduced in Sect. 3.3 and also depicted in Fig. 3.13 as part of the specification graph. Due to the approximation part of the square root algorithm, actor a_3 requires floating point division. In contrast, actor a_2 requires floating point multiplication to check if the error bound is already satisfied. Considering the architecture graph, a Central Processing Unit (CPU) r_{CPU}, a dedicated hardware accelerator r_{HW} for actor a_3, a memory r_{MEM}, and two buses r_{P2P} and r_{BUS} can be identified. Let us assume that



```
class SqrRoot: public smoc_graph {
protected:
  Src a1; SqrLoop a2; Approx a3;
  Dup a4; Sink a5;
public:
  SqrRoot(sc_module_name name)
  : smoc_graph(name), … {…}
};
```

Fig. 3.13 Overview of a codesign flow using SysteMoC as input language for design space exploration of hardware/software alternatives as well as for subsequent code generation and hardware synthesis of each actor called SYSTEMCODESIGNER [15, 25]

r_{CPU} is a slow CPU that has no hardware support for floating point calculations. Hence, floating point calculations must be emulated in software. Thus, while all actors can be mapped to the CPU (see Fig. 3.14a), the actor a_3 will perform significantly worse on the CPU as compared to an implementation alternative (see Fig. 3.14b) where it is mapped to its dedicated hardware accelerator r_{HW} that is connected to the CPU via the point-to-point link r_{P2P}. Finally, the memory r_{MEM}, which is connected by the bus r_{BUS} to the CPU, is used to provide the program memory required by the CPU to implement the actors bound to the CPU as well as the buffer memory required by the channels of the network graph.

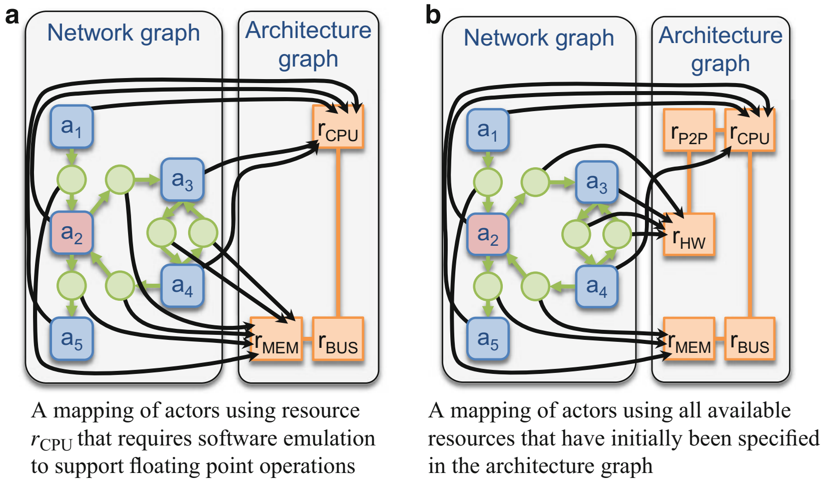

a
Network graph
Architecture graph
a_1
a_2
a_3
a_4
a_5
r_{CPU}
r_{MEM}
r_{BUS}
A mapping of actors using resource r_{CPU} that requires software emulation to support floating point operations
b
Network graph
Architecture graph
a_1
a_2
a_3
a_4
a_5
r_{P2P}
r_{CPU}
r_{HW}
r_{MEM}
r_{BUS}
A mapping of actors using all available resources that have initially been specified in the architecture graph

Fig. 3.14 Depicted above are two possible hardware/software implementation alternatives for Newton's iterative square root algorithm. (**a**) A mapping of actors using resource r_{CPU} that requires software emulation to support floating point operations. (**b**) A mapping of actors using all available resources that have initially been specified in the architecture graph

As discussed above, the network graph of a specification can now be implemented in various ways (cf. Fig. 3.14). In DSE, different mappings of actors and communication channels to physical resources are explored. Resources of the architecture graph not being a target of a mapping are eliminated. As the mapping is independent of the actual MoC, any SysteMoC application can be explored during DSE. The DSE methodology tackles the problem of the exponential explosion of the number of ways the desired functionality can be implemented in an embedded system by performing an automatic optimization of the implementation of the functionality. For a more comprehensive overview of the DSE techniques supported within SYSTEMCODESIGNER, the interested reader is referred to ▶ Chap. 7, "Hybrid Optimization Techniques for System-Level Design Space Exploration".

Moreover, the optimization needs a way to compare implementations with each other. Within SYSTEMCODESIGNER, a simulative performance analysis is performed for SysteMoC applications. For this purpose, a simulative evaluator [38] is realized by parameterizing [39, 43] a given SysteMoC application in such a way that it conforms to the design decisions taken by the DSE. This is established using run-time configurability via the VPC-Plugin [39] or MAESTRO-Plugin [36] in order to model the design decisions taken during DSE. As performance estimation is done by discrete-event simulation, only average case performance can be assessed during exploration for general SysteMoC models. Yet, if the explored network graph is substituted by less expressive models, e.g., by a static data-flow graph, a simple task graph, etc., then also formal model-based analysis techniques for timing, performance, and other objectives of interest such as reliability may be selected and added; see the box analytical evaluators in Fig. 3.13.

Finally, a SysteMoC application may serve as a golden model from which *virtual prototypes* for selected implementations can be derived as well via usage of a *synthesis back-end* (Step 7 in Fig. 3.13) for an implementation alternative selected by the designer (Step 6) from the output of the automatic optimization, i.e., the set of non-dominated or even Pareto-optimal implementations. During synthesis, actors are refined (compiled, synthesized) according to their binding. For this purpose, SystemCoDesigner currently supports hardware synthesis based on Cadence's Cynthesizer. The communication is refined with respect to the chosen model of communication, i.e., shared memory or message passing. So far [15], only shared memory communication using shared address spaces and point-to-point communications using hardware queue implementations are supported in the SystemCoDesigner synthesis back-end.

An important aspect of the SystemCoDesigner flow not shown in Fig. 3.13 is the exploitation of the analysis capability of SysteMoC applications. Although dynamic SysteMoC models can be automatically optimized and assessed by simulation, an additional optimization can be performed for static actors (cf. Sect. 3.5). SystemCoDesigner supports [37] the automatic *clustering* of subgraphs consisting of static actors. Clustering is a transformation where a subgraph of static actors is replaced by a single composite actor implementing a *quasi-static schedule* of the clustered static actors [17, 18]. Such a transformation can reduce the scheduling overhead resulting from dynamic scheduling significantly. For more information about clustering cf. [9].

3.7 Conclusions

Data-flow Models of Computation (MoCs) are well suited for the modeling of many applications that are targeted to heterogeneous hardware/software systems. Due to the inherently concurrent behavior of actors, partitioning into hardware and software blocks can be done at the granularity of actors. It is our recommendation therefore to think in terms of actors when designing a system because of the natural concurrency available in these models. However, there is a trade-off between expressiveness and analyzability of different data-flow MoCs. In this chapter, different data-flow models have been presented in a consistent framework. Moreover, SysteMoC, a MoC with a very high expressiveness, has been discussed. Its most outstanding property is the option to automatically identify if a SysteMoC actor uses this expressiveness or if it can be classified to belong to a more restricted data-flow MoC. In the latter case, this knowledge increases the analyzability. This advantage may be greatly exploited also during later optimization phases as well as during code generation and hardware synthesis from actor specifications to final hardware/software system implementations.

References

1. Bhattacharya B, Bhattacharyya SS (2001) Parameterized dataflow modeling for DSP systems. IEEE Trans Signal Process 49(10):2408–2421. doi:10.1109/78.950795
2. Bhattacharyya SS, Murthy PK, Lee EA (1997) APGAN and RPMC: complementary heuristics for translating DSP block diagrams into efficient software implementations. J Des Autom Embed Syst 2:33
3. Bilsen G, Engels M, Lauwereins R, Peperstraete J (1996) Cyclo-static dataflow. IEEE Trans Signal Process 44(2):397–408
4. Blickle T, Teich J, Thiele L (1998) System-level synthesis using evolutionary algorithms. Des Autom Embed Syst 3(1):23–58
5. Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D dissertation, Department of EECS, University of California, Berkeley. Technical Report UCB/ERL 93/69
6. Commoner F, Holt AW, Even S, Pnueli A (1971) Marked directed graphs. J Comput Syst Sci 5(5):511–523. doi:10.1016/S0022-0000(71)80013-2
7. Dennis J (1974) First version of a data flow procedure language. In: Robinet B (ed) Programming symposium. Lecture notes in computer science, vol 19. Springer, Berlin/Heidelberg, pp 362–376. doi:10.1007/3-540-06859-7_145
8. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity – the ptolemy approach. Proc IEEE 91(1):127–144. doi:10.1109/JPROC.2002.805829
9. Falk J (2015) A clustering-based MPSoC design flow for data flow-oriented applications. Dr. Hut, Sternstr. 18, München. Dissertation Friedrich-Alexander-Universität Erlangen-Nürnberg
10. Falk J, Haubelt C, Teich J (2005) Syntax and execution behavior of SysteMoC. Technical report. 04-2005, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, D-91058 Erlangen
11. Falk J, Haubelt C, Teich J (2006) Efficient representation and simulation of model-based designs in systemC. In: Proceedings of the forum on specification and design languages (FDL 2006), pp 129–134
12. Falk J, Haubelt C, Zebelein C, Teich J (2013) Integrated modeling using finite state machines and dataflow graphs. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of signal processing systems. Springer, Berlin/Heidelberg, pp 975–1013
13. Falk J, Keinert J, Haubelt C, Teich J, Bhattacharyya SS (2008) A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In: Proceedings of the 8th ACM international conference on embedded software (EMSOFT 2008). ACM, New York, pp 189–198. doi:10.1145/1450058.1450084
14. Falk J, Schwarzer T, Glaß M, Teich J, Zebelein C, Haubelt C (2015) Quasi-static scheduling of data flow graphs in the presence of limited channel capacities. In: Proceedings of the 13th IEEE symposium on embedded systems for real-time multimedia (ESTIMEDIA 2015) p 10
15. Falk J, Schwarzer T, Zhang L, Glaß M, Teich J (2015) Automatic communication-driven virtual prototyping and design for networked embedded systems. Microprocess Microsyst 39(8):1012–1028. doi:10.1016/j.micpro.2015.08.008
16. Falk J, Zebelein C, Haubelt C, Teich J (2011) A rule-based static dataflow clustering algorithm for efficient embedded software synthesis. In: Proceedings of the design, automation and test in Europe (DATE 2011). IEEE, pp 521–526
17. Falk J, Zebelein C, Haubelt C, Teich J (2013) A rule-based quasi-static scheduling approach for static Islands in dynamic dataflow graphs. ACM Trans Embed Comput Syst 12(3):74:1–74:31. doi:10.1145/2442116.2442124

18. Falk J, Zebelein C, Keinert J, Haubelt C, Teich J, Bhattacharyya, SS (2011) Analysis of systemC actor networks for efficient synthesis. ACM Trans embed Comput Syst 10(2):18:1–18:34. doi:10.1145/1880050.1880054
19. Girault A, Lee B, Lee E (1999) Hierarchical finite state machines with multiple concurrency models. IEEE Trans Comput Aided Des Integr Circuits Syst 18(6):742–760
20. Gouda MG (1980) Liveness of marked graphs and communication and VLSI systems represented by them. Technical report, Austin
21. Gu R, Janneck JW, Raulet M, Bhattacharyya SS (2011) Exploiting statically schedulable regions in dataflow programs. Signal Processing Syst 63(1):129–142. doi:10.1007/s11265-009-0445-1
22. Hsu CJ, Bhattacharyya SS (2007) Cycle-breaking techniques for scheduling synchronous dataflow graphs. Technical report. UMIACS-TR-2007-12, Institute for Advanced Computer Studies, University of Maryland at College Park
23. Hylands C, Lee E, Liu J, Liu X, Neuendorffer S, Xiong Y, Zhao Y, Zheng H (2004) Overview of the ptolemy project, technical memorandum no. UCB/ERL M03/25. Technical report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
24. Kahn G (1974) The semantics of a simple language for parallel programming. In: IFIP Congress, pp 471–475
25. Keinert J, Streubühr M, Schlichter T, Falk J, Gladigau J, Haubelt C, Teich J, Meredith M (2009) SystemCoDesigner – an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. Trans Des Autom Electron Syst 14(1):1:1–1:23
26. Kosinski PR (1978) A straightforward denotational semantics for non-determinate data flow programs. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL 1978). ACM, New York, pp 214–221. doi:10.1145/512760.512783
27. Lee EA (2006) The problem with threads. Technical report. UCB/EECS-2006-1, EECS Department, University of California, Berkeley. The published version of this paper is in IEEE Computer 39(5):33–42, May 2006
28. Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans Comput 36(1):24–35. doi:10.1109/TC.1987.5009446
29. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
30. Lee EA, Sangiovanni-Vincentelli AL (1998) A framework for comparing models of computation. IEEE Trans Comput Aided Des Integr Circuits Syst 17(12):1217–1229. doi:10.1109/43.736561
31. Parks TM (1995) Bounded scheduling of process networks. Ph.D dissertation, Department of EECS, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/1995/2926.html. Technical Report UCB/ERL M95/105
32. Pino JL, Bhattacharyya SS, Lee E (1995) A hierarchical multiprocessor scheduling system for DSP applications. In: Proceedings of the Asilomar conference on signals, systems, and computers, vol 1, pp 122–126. doi:10.1109/ACSSC.1995.540525
33. Plishker W, Sane N, Kiemb M, Anand K, Bhattacharyya SS (2008) Functional DIF for rapid prototyping. In: The 19th IEEE/IFIP international symposium on rapid system prototyping (RSP 2008), pp 17–23. doi:10.1109/RSP.2008.32
34. Plishker W, Sane N, Kiemb M, Bhattacharyya SS (2008) Heterogeneous design in functional DIF. In: Proceedings of the 8th international workshop on embedded computer systems: architectures, modeling, and simulation (SAMOS 2008). Springer, Berlin/Heidelberg, pp 157–166. doi:10.1007/978-3-540-70550-5_18
35. Ptolemaeus C (ed) (2014) System design, modeling, and simulation using Ptolemy II. Ptolemy.org, Berkeley. http://ptolemy.org/systems
36. Rosales R, Glaß M, Teich J, Wang B, Xu Y, Hasholzner R (2014) Maestro – holistic actor-oriented modeling of nonfunctional properties and firmware behavior for mpsocs. ACM Trans Des Autom Electron Syst (TODAES) 19(3):23:1–23:26. doi:10.1145/2594481

37. Schwarzer T, Falk J, Glaß M, Teich J, Zebelein C, Haubelt C (2015) Throughput-optimizing compilation of dataflow applications for multi-cores using quasi-static scheduling. In: Stuijk S (ed) Proceedings of the 18th international workshop on software and compilers for embedded systems (SCOPES 2015). ACM, Berlin, pp 68–75
38. Streubühr M, Falk J, Haubelt C, Teich J, Dorsch R, Schlipf T (2006) Task-accurate performance modeling in systemC for real-time multi-processor architectures. In: Gielen GGE (ed) Proceedings of design, automation and test in Europe (DATE 2006). European Design and Automation Association, Leuven, pp 480–481. doi:10.1145/1131610
39. Streubühr M, Gladigau J, Haubelt C, Teich J (2010) Efficient approximately-timed performance modeling for architectural exploration of MPSoCs. In: Borrione D (ed) Advances in design methods from modeling languages for embedded systems and SoC's. Lecture notes in electrical engineering, vol 63. Springer, Berlin/Heidelberg, pp 59–72. doi:10.1007/978-90-481-9304-2_4
40. Stuijk S, Geilen M, Theelen BD, Basten T (2011) Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications. In: Proceedings of the international conference on embedded computer systems: architectures, modeling, and simulation (ICSAMOS 2011). IEEE Computer Society, pp 404–411. doi:10.1109/SAMOS.2011.6045491
41. Thiele L, Strehl K, Ziegenbein D, Ernst R, Teich J (1999) FunState – an internal design representation for codesign. In: White JK, Sentovich E (eds) ICCAD. IEEE, pp 558–565
42. Thiele L, Teich J, Naedele M, Strehl K, Ziegenbein D (1998) SCF – state machine controlled flow diagrams. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Gloriastrasse 35, CH-8092. Technical Report TIK-33
43. Xu Y, Rosales R, Wang B, Streubühr M, Hasholzner R, Haubelt C, Teich J (2012) A very fast and quasi-accurate power-state-based system-level power modeling methodology. In: Herkersdorf A, Römer K, Brinkschulte U (eds) Proceedings of the architecture of computing systems (ARCS 2012), vol 7179. Springer, Berlin/Heidelberg, pp 37–49. doi:10.1007/978-3-642-28293-5_4
44. Zebelein C (2014) A model-based approach for the specification and refinement of streaming applications. Ph.D. thesis, University of Rostock
45. Zebelein C, Falk J, Haubelt C, Teich J (2008) Classification of general data flow actors into known models of computation. In: Proceedings 6th ACM/IEEE international conference on formal methods and models for codesign (MEMOCODE 2008), pp 119–128