SN Computer Science (2022) 3:249
https://doi.org/10.1007/s42979-022-01102-3

ORIGINAL RESEARCH

Synthesis of Parallel Software from Heterogeneous Dataflow Models

Omair Rafique[1] · **Klaus Schneider**[1]

Received: 2 October 2021 / Accepted: 21 March 2022 / Published online: 26 April 2022

Abstract
Dataflow process networks (DPNs) are intrinsically data-driven, i.e., node actions are not synchronized among each other and may fire whenever sufficient input operands arrived at a node. While the general model of computation (MoC) of DPNs does not impose further restrictions, many different subclasses of DPNs representing different dataflow MoCs have been considered over time. These classes mainly differ in the kinds of behaviors of the processes. A DPN may be heterogeneous in that different processes in the network belong to different classes of DPNs. A heterogeneous DPN can therefore be effectively used to model and to implement different components of a system with different kinds of processes and, therefore, different dataflow MoCs. This paper presents a model-based design based on different dataflow MoCs including their heterogeneous combinations. In particular, it covers the automatic software synthesis of systems from DPN models. The main objective is to validate, evaluate and compare the artifacts exhibited by different dataflow MoCs at the implementation level of systems under the supervision of a common design tool. Moreover, this work also offers an efficient synthesis method that targets and exploits heterogeneity in DPNs by generating implementations based on the kinds of behaviors of the processes. The proposed synthesis method provides a tool chain including different specialized code generators for specific dataflow MoCs, and a runtime system that finally maps models using a combination of different dataflow MoCs on cross-vendor target hardware.

Keywords Dataflow process networks · Heterogeneity · Automatic synthesis

Introduction

The State of the Art

In a model-based design of embedded systems, we start by focusing on models that describe the functional behavior of the system. These models are hardware independent and are based on a particular model of computation (MoC). A MoC determines *why, when, which atomic action of a system is executed*. A model-based design is typically equipped with a tool chain that following a correctness-by-construction approach finally produces executable code. The code can then be deployed on the target hardware such as a CPU, a GPU or even an FPGA depending on the target application. Depending on the application as well as on the target architecture, different MoCs have their own advantages and disadvantages.

For example, when it comes to real-time systems, the synchronous reactive (SR) [5, 6, 43] MoC greatly simplifies many efforts in the validation and verification. In fact, it has proven its usefulness both on single-core and multi-core platforms, as well as on application-specific hardware platforms in safety critical applications such as avionics [12] and other embedded system industries. However, when it comes to soft real-time applications such as streaming and signal processing [28], performance and design flexibility are often dominant factors over safety, and commercial off-the-shelf (COTS) heterogeneous hardware platforms are preferred [2]. The generation of distributed implementations is often desired for such applications where different components are mapped and executed on different computing units (devices). For such applications, especially when implemented on heterogeneous platforms, synchronization and communication

This article is part of the topical collection "Model-Driven Engineering and Software Development" guest edited by Slimane Hammoudi and Luis Ferreira Pires.

✉ Omair Rafique
rafique@cs.uni-kl.de

Klaus Schneider
schneider@cs.uni-kl.de

[1] Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany

overheads caused by synchronous semantics often reduce the performance [1].

Instead, asynchronous models or dataflow process networks (DPNs) [11, 22, 23] are well-suited for the implementation of such systems. In particular, they explicitly expose the concurrency of applications and thereby simplify their execution on parallel and heterogeneous architectures. However, it is not beneficial to start with DPNs in model-based designs since DPNs do not lend themselves well for simulation and verification. In particular, major correctness properties like buffer boundedness and absence of deadlocks are not decidable for general DPNs [18, 29]. Therefore, both SR and DPNs have their own advantages and disadvantages.

As an alternative, *desynchronization* of synchronous models [4, 19] has been developed that benefits both from the static analysis methods for synchronous systems and the performance of the finally synthesized asynchronous systems. Desynchronization techniques [1, 4, 31] are used to convert synchronous models into asynchronous ones to more efficiently support distributed implementations. These techniques preserve the functional specification of the synchronous models and, moreover, preserve properties like deadlock-freedom and bounded memory usage that are otherwise difficult to ensure in DPNs. These desynchronized models [1, 3] are the *starting point* of this work. The resulting dataflow models are free from deadlocks and even buffers with only single entries are sufficient for a successful execution.

Motivation and Problem Setting

DPNs consist of statically defined process nodes with first-in-first-out (FIFO) buffered point-to-point connections. The process nodes are not synchronized among each other and may fire whenever sufficient data is available for a node. While the general MoC of DPNs does not impose further restrictions, many different classes of DPNs have been considered over time like Kahn process networks (KPN) [22], (cyclo-)static dataflow (SDF) [30] networks and Boolean dataflow (BDF) [10] networks. Each class defines a specific dataflow MoC by specifying a particular execution and communication semantics [16]. These classes mainly differ in the kinds of behaviors of the processes which affect on the one hand the expressiveness of the DPN class as well as the methods for their analysis (predictability) and synthesis (efficiency). These behaviors are precisely described based on the underlying semantics of how each process is triggered for an execution, and how each execution of a process consumes/produces data. A process in a static DPN exhibits a static behavior where a statically determined amount of data is consumed and produced in each execution. Second, a process in a KPN has a sequential behavior where a dynamically determined amount of data is consumed and produced sequentially in each execution. In contrast, a process in a dynamic dataflow (DDF) network may have a parallel behavior that can consume and produce a dynamically determined amount of data in parallel. A DPN may be heterogeneous in that different processes in the network may belong to different classes of DPNs. A heterogeneous combination of particular kinds of processes can be used to model and implement different components of a system with different kinds of processes and, therefore, different dataflow MoCs. A simple example of a heterogeneous DPN consisting of different kinds of behaviors of the processes is visualized in Fig. 1.

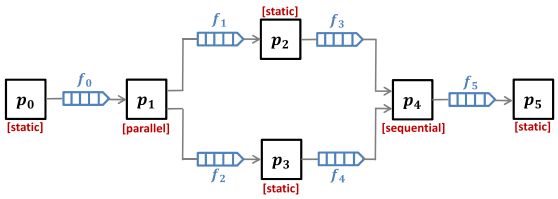

Fig. 1 A simple visualization of a heterogeneous DPN. It consists of different kinds of processes $(p_0, \ldots, p_5)$ characterized by static, sequential and parallel behaviors. The processes $(p_0 \ldots p_5)$ are connected together via FIFO buffered point-to-point channels $(f_0, \ldots, f_5)$

Design tools for modeling [9, 14, 20, 24, 42, 49] are used to model and to design parallel embedded systems using certain MoCs, including different dataflow MoCs. However, there is a lack of automatic synthesis methods to analyze and to evaluate the artifacts exhibited by particular MoCs. Second, the existing design tools for synthesis like [8, 28, 45] are usually restricted to the weakest classes of DPNs, i.e., to cyclo-static and static DPNs where each tool only supports a specific dataflow MoC. For heterogeneous DPNs offering heterogeneous combinations of different kinds of behaviors of the processes, the synthesis method should exploit this heterogeneity by generating efficient implementations based on the dataflow MoC of each process.

Apart from efficiency, another crucial challenge is the portability of applications on different cross-vendor platforms which is not systematically handled by the traditional design flows. In general, a non-trivial manual effort is finally required for deploying automatically generated code to a particular target architecture.

The overall motivation of this work is to enable the automatic software synthesis of systems using different dataflow MoCs including their heterogeneous combinations. The main objective is to validate, evaluate and compare the artifacts exhibited by different dataflow MoCs at the implementation level of systems under the supervision of a common design tool. Moreover, the idea is to offer an efficient synthesis method that exploits heterogeneity in dataflow networks by generating implementations based on the kinds of behaviors of the processes. Finally, this work also considers the challenge of systematically handling the portability of

modeled systems on cross-vendor heterogeneous platforms as an integral part of the synthesis process.

Contributions

We propose a synthesis design flow that essentially enables the automatic software synthesis of systems based on different dataflow MoCs. In particular, it supports three different dataflow MoCs, namely synchronous (static) dataflow (SDF) [26], Kahn process networks (KPN) [21], and a deterministic variant of dynamic dataflow (DDF). The common design tool can be effectively used to generate implementations based on the individual dataflow MoCs [32, 35, 41]. Moreover, in contrast to existing dataflow oriented synthesis methods, the proposed method efficiently targets and exploits heterogeneity in dataflow networks by generating implementations purely based on the kinds of behaviors of the processes or the underlying dataflow MoC of each process [36].

The target DPN model of our desynchronization method is based on a limited subset of the Cal actor language (CAL) [13]. The proposed synthesis design flow provides a comprehensive tool chain, including different specialized code generators for specific dataflow MoCs, and a runtime system that finally maps models using a combination of different dataflow MoCs on the target hardware. The tool chain essentially offers a platform-independent code synthesis method based on the open computing language (OpenCL) [47] abstraction that enables a more generalized synthesis targeting COTS heterogeneous architectures. In particular, this work focuses on mapping modeled systems on cross-vendor multi-core CPUs and many-core GPUs.

The main contributions of this work can be summarized as follows:

- We propose an automatic model-based synthesis that allows us to synthesize systems using different dataflow MoCs, namely the SDF MoC, the KPN MoC and a deterministic variant of the DDF MoC.
- We implemented a platform-independent code synthesis method for CAL DPN models. In particular, we offer a synthesis tool chain that automatically synthesizes CAL models into platform-independent OpenCL code.
- We offer a single back-end based on OpenCL which is comprised of different specialized code generators for specific dataflow MoCs.
- We present the runtime system designed under the OpenCL abstraction for finally deploying DPNs on cross-vendor COTS target hardware.

Related Work

A number of model-based design tools have been presented over time for the design and development of embedded systems. This section covers a number of well-established design tools, categorized mainly from the perspective of desired goals, employed strategies and usage as given in the following sections.

Design Tools for Modeling

The Ptolemy project [9, 14] is a design tool originally constructed in a Java-based environment to support the modeling and simulation of behaviors based on different MoCs, including particular dataflow MoCs. Although the main focus is to study and analyse different MoCs at the modeling level, it also provides preliminary code generators.[1] It requires a supporting helper code for each process which is provided manually using a fairly complex procedure. FERAL [24] is another framework developed to provide a holistic model-based design approach to enable the coupling of specialized simulators in offline scenarios, i.e., without connecting them to real hardware. This project very interestingly adopts and extends the concepts from the Ptolemy project.

The formal system design (ForSyDe) [42] tool offers a formal design methodology for embedded systems based on different MoCs including the SR MoC and two particular dataflow MoCs. Although the major focus of this design tool is the modeling framework, it also provides a hardware synthesis tool that has been mainly elaborated for translating models limited to the SR MoC into the corresponding VHDL code. Another synthesis plug-in called *f2cc*[2] has been introduced for generating GPGPU software code from models limited to the SR MoC.

The SystemC models of computation (SysteMoC) [20] is an actor-oriented dataflow programming language built on top of SystemC. Besides supporting different dataflow MoCs, it also offers the automatic MoC identification of processes (actors), which is not featured in frameworks like Ptolemy and ForSyDe. Although the main focus of SysteMoC has been at the design level, the System-CoDesigner [20] framework specializes in automatic design space exploration starting from SysteMoC models. In particular, the framework offers a platform-based automatic system generation from SysteMoC models.

SDF for free (SDF^3) [49] is a versatile experimental tool that can generate random static dataflow graphs (SDFGs)

[1] http://ptolemy.berkeley.edu/ptolemyII/ptII10.0/ptII10.0.1/ptolemy/cg/.

[2] https://github.com/forsyde/f2cc/wiki.

with support to analyse and visualize these graphs. It supports three different classes of static DPNs, namely the static dataflow (SDF) [26], the cyclo-static dataflow (CSDF) [15] and the scenario-aware dataflow (SADF) [48]. The tool includes an extensive library of SDFG analysis and transformation algorithms as well as functionality to visualize and simulate them.

Design Tools for Synthesis

Model-based design tools for synthesis in the related state-of-the-art mainly differ by their employed MoCs. A number of dataflow-oriented design tools have been presented where each tool usually only supports a specific dataflow MoC. To this end, some of the inspiring model-based design tools for synthesis are presented in [8, 28, 44, 45] (to name a few).

The framework presented in [45] introduces a design flow for executing applications specified as SDF graphs on heterogeneous systems using OpenCL. However, it only supports the execution of behaviors limited to SDF.

The work presented in [28] translates DPNs modeled using a subset of CAL to parallel programs based on OpenCL. The methodology incorporates static analysis and transformations and thus confined to the synthesis of SDF models. Similarly, the dataflow oriented framework [8] proposes a dataflow MoC as a symmetric-rate dataflow, a variant of SDF where the token production rate and the token consumption rate per FIFO channel is symmetric.

The distributed application layer (DAL) framework [44] presents a scenario-based design flow for mapping streaming applications onto heterogeneous on-chip many-core systems. Behaviors are modeled based on a specific dataflow MoC, namely the KPN MoC [22], and the execution scenarios are coordinated using a finite state machine (FSM).

Design Tools Used in Industry

One of the most popular and commercially recognized model-based design tool Matlab[3] has introduced a variety of supporting toolkits over time. Interestingly, Matlab Simulink introduced the dataflow domain[4] where applications can be modeled and simulated based on the SDF MoC. The main objective of introducing the dataflow domain is to improve the simulation throughput with multithreaded execution.

The Signal Processing Worksystem (SPW) from Cadence Design Systems[5] supports the modeling and analysis of signal processing algorithms based on static as well as dynamic dataflow models. The design flow mainly focuses on the simulation and manual refinement of modeled systems. Similarly, CoCentric System Studio from Synopsys[6] is a system-level design solution consisting of tools, methodologies, and libraries that enables the design and simulation of systems-on-a-chip. The modeling paradigms can be hierarchically mixed at all levels for e.g., based on nested dataflow models and FSMs. The main emphasis of the design flow is the modeling and analysis of complex systems.

Tools	Multiple dataflow MoCs	Automatic synthesis	Heterogeneous synthesis	Heterogeneous target hardware
Ptolemy [9, 14]	✓	×	×	×
SysteMoC [20]	✓	✓	×	×
FERAL [24]	✓	×	×	×
SDF³ [48]	✓	×	×	×
Schor et al. [45]	×	✓	×	✓
PRUNE [8]	×	✓	×	✓
DAL [44]	×	✓	×	×
Lund et al. [28]	×	✓	×	✓
Proposed tool	✓	✓	✓	✓

Fig. 2 Comparison of the proposed framework with related tools

Summary

In general, model-based design tools for embedded systems that support heterogeneous combinations of MoCs including different dataflow MoCs are of particular interest for modeling and analysis of complex systems. These frameworks, developed and evolved over decades, are used to formally analyse different MoCs for modeling and designing embedded systems. Some of the design tools in this category also introduced a synthesis facility, supporting platform-dependent synthesis methods usually restricted to implementations based on particular MoCs. In general, there is a lack of automatic synthesis methods to analyse and to evaluate different dataflow MoCs. Second, the existing design tools for synthesis are usually dedicated to automatically implementing systems based on a specific dataflow MoC. Therefore, a common synthesis design flow is still needed that mainly focuses and emphasizes on the automatic software synthesis of systems based on different dataflow MoCs. Moreover, an efficient software synthesis method is desired that targets and essentially exploits heterogeneity in dataflow networks by generating implementations based on the dataflow MoC of each process. The features offered by the proposed tool in comparison to related tools are summarized in Fig. 2.

[3] http://www.mathworks.com/matlabcentral/.

[4] https://www.mathworks.com/help/dsp/ug/dataflow-domains.html/.

[5] https://www.cadence.com/.

[6] https://www.synopsys.com/.

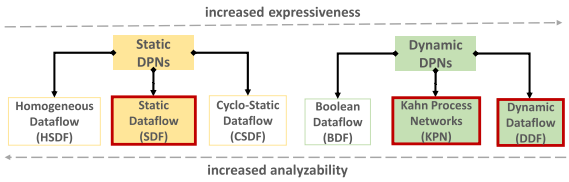

Fig. 3 Categorization of various dataflow MoCs. Different dataflow MoCs are categorized into static and dynamic ones. The static DPNs are ones having fixed consumption and production rates. Whereas, the dynamic DPNs involve variable consumption and production rates. The analyzability of DPNs is inversely related to their expressiveness. The dataflow MoCs supported in this work are highlighted by colored boxes

Preliminaries

This section highlights the background of this work by presenting some important preliminaries. This includes the tools and specifications used as essential parts of the proposed design flow.

Dataflow Process Networks

A dataflow process network (DPN) [11, 22, 23] describes the behavior of a system by distributing it in a set of independent process nodes that interact with each other only through FIFO buffered point-to-point channels, as shown in Fig. 1. Each process performs a computation by firing where it consumes data tokens from its input buffers and produces data tokens for its output buffers. The behavior of each process is described by firing rules which are triggered by the availability of data. The general MoC of DPNs does not impose further restrictions. However, a number of different classes of DPNs representing different dataflow MoCs have been considered over time [16]. These classes mainly differ in the kinds of behaviors of the processes. These behaviors are precisely described based on the underlying semantics of how each atomic process is triggered for an execution, and how each execution of a process consumes/produces data, in particular, whether a statically or dynamically determined amount of data is consumed and produced. Based on that, the most commonly known classes can be categorized into *static* and *dynamic* DPNs as depicted in Fig. 3.

The latter accommodates DPNs like Kahn process networks (KPN) [22], Boolean dataflow (BDF) [10] and the dynamic dataflow (DDF) networks. Whereas, the former includes DPNs like static dataflow (SDF) [26, 30], homogeneous synchronous dataflow (HSDF) [26] and the cyclo-static dataflow (CSDF) [15] networks. Static DPNs are generally characterized as having only processes where the consumption and production of tokens are neither influenced by the values of the consumed tokens nor are they dependent on the points in time at which tokens arrive on the input buffers. Thus, processes in static DPNs always consume the same number of input tokens from particular input buffers and produce the same number of output tokens to particular output buffers. However, they may read different number of tokens from different input buffers and may write different number of tokens to different output buffers. On the one hand, these characteristics allow powerful design-time analysis techniques (e.g., for performance analysis and verification), but on the other hand, they limit the expressiveness by excluding dynamic behaviors (like select and switch nodes).

In contrast to static DPNs, processes in dynamic DPNs can vary the consumption and production of tokens in each firing dependent on the history of the consumed tokens and also on the tokens to be consumed. This allows conditional or data-dependent executions of processes; in particular, each process can produce and consume a different number of tokens in every firing. This generalization results in higher expressiveness and flexibility but makes the analysis more difficult.

In general, DPNs offer a modeling paradigm well suited for the modeling of concurrent embedded systems. However, model-based designs starting with dynamic DPNs have to deal with analyzability issues, i.e., the undecidability of checking major correctness properties like buffer boundedness and absence of deadlocks. Therefore, implementations of concurrent and distributed embedded systems from DPNs like KPNs may suffer from problems like deadlocks and buffer overflows [18, 29].

Desynchronized DPN Model

As a long-term project, our group developed the *Averest*[7] tool for a model-based design process starting with synchronous models. The *Averest* project aims at providing a complete set of tools for the development of reactive systems. Moreover, the work presented in [1, 3] further presents a desynchronization design flow based on *Averest*. The complete design flow based on *Averest* is presented in [2]. Since synchronous models are particularly well suited for analysis, the design flow starts with synchronous models, verifies them for desynchronization and then translates them to DPNs for the synthesis of concurrent and distributed systems. The underlying language of the target DPN model is a limited subset of CAL. Since, the proposed synthesis method targets the execution and deployment of DPNs on heterogeneous platforms consisting of different types of devices including GPUs, the desynchronization method generates stateless dataflow processes. This simplifies not only the target DPN specification for the final synthesis, but also paves the way for dynamically handling parallelization in OpenCL

[7] http://www.averest.org.

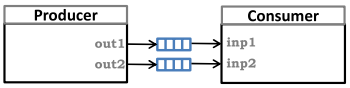

Fig. 4 A simple example of a producer-consumer network

based synthesized implementations. Moreover, because of the great similarity between synchronous guarded actions (SGAs) as the *Averest* intermediate format (AIF) and CAL guarded actions, the correctness of the translation method is easily verified.

The target subset of CAL, therefore, simply consists of a set of guarded actions. Thereby, each generated stateless process essentially consists of a set of guarded actions where the guards are applied to the values of the input tokens. Depending on the behavior of a particular synchronous module, the generated process possesses a particular kind of behavior that precisely determines a particular dataflow MoC. To exemplify, CAL processes based on different supported dataflow MoCs are illustrated in Figs. 9, 10, and 11.

Apart from processes, the topology of the network is usually described using the functional network language (FNL) based on the XML format [7]. A simple producer-consumer dataflow network, as shown in Fig. 4, is specified in FNL as shown in Listing 1. This example shows the two most basic elements of FNL, namely the *Instance* and the *Connection*. Each *Instance* field defines a process instance (Lines 2–4 and 5–7), and possibly can even refer to another network. Each *Connection* field defines a connection between an input port and an output port of two instances (Lines 8–11).

Hence, the generated desynchronized CAL code consists of two parts: the CAL processes and the network description.

Open Computing Language

The open computing language OpenCL [47] has been designed for parallel computing on cross-vendor and heterogeneous architectures. In contrast to proprietary specification languages with limited hardware choices, OpenCL allows task-parallel and data parallel heterogeneous computing on a heterogeneous collection of modern central processing units (CPUs), graphical processing units (GPUs), digital signal processors (DSPs), and other microprocessor designs organized into a single platform [25, 46].

A primary benefit of OpenCL is a substantial acceleration in parallel processing. OpenCL supports both coarse-grained (task-level) as well as fine-grained (data-level) parallelism. Second, it provides the ability to write vendor-neutral cross platform applications. These benefits can be derived by understanding and exploiting a set of abstract models provided by OpenCL, as depicted in Figs. 5 and 6.

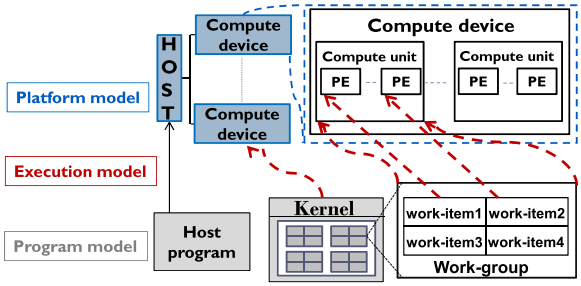

Fig. 5 Overview of the OpenCL architecture: the *platform model* provides a standard abstraction of the target hardware. The *program model* specifies the behavior of a system typically organized as a host and several kernels. The *execution model* describes the mapping of the program model onto the platform model

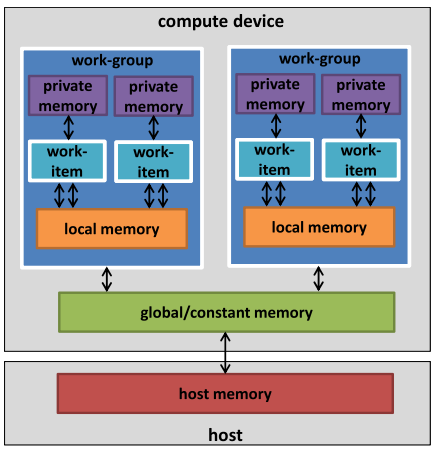

Fig. 6 OpenCL memory model consists of five regions: The *host memory* is only accessible to the host processor. The *global memory* is accessible to both the host and device. The *constant memory* is fully accessible to the host and write-protected for the device. The *local memory* is only visible to the host and is local to a single compute unit. The *private memory* is private to an individual work-item executing within an OpenCL processing element https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html

```
1  <XDF name="ConProd">
2      <Instance id="producer">
3          <Class name="cal.producer"/>
4      </Instance>
5      <Instance id="consumer">
6          <Class name="cal.consumer"/>
7      </Instance>
8      <Connection dst="consumer" dst_port="inp1"
9      src="producer" src_port="out1"/>
10     <Connection dst="consumer" dst_port="inp2"
11     src="producer" src_port="out2"/>
12 </XDF>
```

Listing 1 The FNL network description for the producer-consumer example as shown in Fig. 4.

Platform Model

The OpenCL platform model provides users with a convenient abstraction of the target hardware. It is defined as a *host* connected to one or more compute devices, each having multiple compute units (CUs), each of which further consists of multiple processing elements (PEs).

A host is typically a CPU running a standard operating system (OS), while a compute device may be a GPU, a DSP, a further multicore CPU or any other specific microprocessor. Each device, therefore, consists of a collection of one or more CUs where each CU can be conceived as, for instance, a core of a CPU, or a streaming multiprocessor of a GPU. A CU is further composed of one or more PEs that execute instructions. Each PE can, therefore, be conceived as, for instance, a streaming core (or SIMD lane) of a GPU. An OpenCL device, therefore, executes the instruction computations on the PEs within the device.

Program Model

The OpenCL program model is comprised of two main components: the *host program* and *kernels*. The host program executes on the host, defines device contexts, sets up command queues of devices and enqueues instances of kernel executions on devices.

Kernels: A kernel is a C-like function that actually implements the abstract behavior of the system or part of the system. OpenCL targets the parallel execution of a kernel on compute devices by organizing it into a computation domain. This computation domain is defined when a kernel is mapped for execution on the command queue. Each independent element of this domain represents the execution instance of the kernel and is termed as the *work-item*. Each work-item performs the same kernel function but on different data. OpenCL also allows grouping work-items together into *work-groups*, as shown in Fig. 5. All work-items in the same work-group are executed together on the same compute unit.

Host Program: It resides and executes on the host and is responsible for setting up and handling the execution of kernels on the compute devices using the defined context. The context essentially sets up the environment for executing kernels and is created with a set of devices. After the context is created, command queues are created where the kernels are mapped to get executed on the OpenCL devices associated with the context. Each command queue can represent a complete device (e.g., a CPU) or even a compute unit of that device (e.g., a CPU-core).

Execution Model

The OpenCL execution model can be understood as the mapping of kernels on the platform model which is implemented in the host program. Depending on the target compute device (e.g., a CPU or a GPU), kernels are mapped differently. In case of GPUs, OpenCL only allows the user to create a command queue at the level of a compute device. Hence, for a GPU, a kernel is typically allocated on a compute device, a work-group is ideally mapped on a CU, and work-items of that work-group are executed by PEs of that CU, as depicted in Fig. 5. In contrast, for CPUs, a command queue can be created at the level of a compute device as well as at the level of a CU. For the latter, the whole kernel (all work-groups) are mapped to the same CU (i.e., a core of a CPU).

Memory Model

OpenCL offers a disjoint memory model to programmers as shown in Fig. 6. This is mainly because OpenCL targets heterogeneous platforms where most platforms utilize disjoint memory systems due to different memory requirements of different architectures. OpenCL visualizes its target as a system where data sharing between the host and compute devices is performed explicitly by a system network, such as a peripheral component interconnect (PCI) bus. The OpenCL memory model is organized in five regions consisting of *host*, *global*, *constant*, *local* and *private* memories. For heterogeneous architectures consisting of multiple devices integrated on a single platform, host memory and device memory are independent of one another. This requires the explicit handling of data from host memory to device memory and back to host.

The Design Flow: Overview

The overall design flow can be understood in two parts, i.e., the modeling part and the synthesis part as shown in Fig. 7. In general, the starting point of this work is a desynchronized model. The desynchronization method based on our *Averest* tool finally generates a CAL DPN based on the supported dataflow MoCs. To this end, a general CAL DPN model is considered that relies on an abstract notion of a process. A process is composed of a finite set of actions where each action can perform a computation by consuming tokens from input buffers and producing tokens to output buffers. This general model is used with specific constraints and restrictions to specify the precise dataflow MoCs. Each supported dataflow MoC interprets the behavior of the CAL process in two parts: (i) process triggering or scheduling and (ii) process execution. The triggering behavior determines the conditions under which the dataflow MoC triggers a process

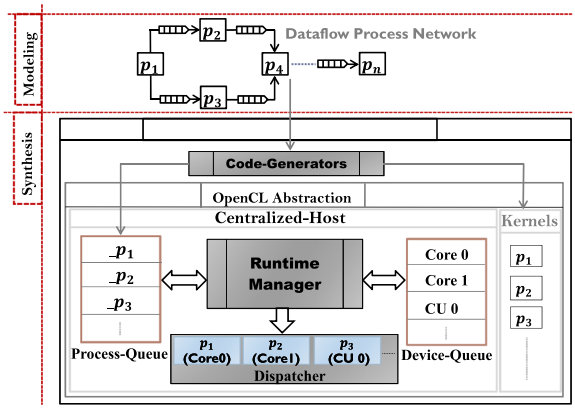

Fig. 7 The basic building block diagram [35] of the proposed framework. It can be understood in two phases: the modeling phase is in general provided with the desynchronized CAL DPN models. The synthesis phase employs the OpenCL abstraction and features a tool chain involving the specialized code generators and the runtime system that finally executes and maps the desynchronized models on the OpenCL abstracted target hardware

for an execution whereas the execution behavior determines how a process consumes/produces data when it is triggered for an execution. The framework supports three different dataflow MoCs, namely the synchronous (static) dataflow (SDF) [26] MoC, the Kahn process networks (KPN) [21] MoC and a deterministic variant of the dynamic dataflow (DDF) MoC. The general DPN model based on the used CAL subset and the supported dataflow MoCs are described in Sect. "Modeling: Dataflow Models". The desynchronized model consisting of CAL processes and the network specification is provided as input to the synthesis phase.

The synthesis part as shown in Fig. 7 provides a comprehensive tool chain, including a single back-end that offers different specialized code generators for different dataflow MoCs, and a runtime system which finally executes DPNs on the target hardware. Using OpenCL [47], it incorporates a standard hardware abstraction for cross-vendor heterogeneous hardware architectures. The proposed framework conceptually employs OpenCL as an operating system (OS) in the sense that it provides: common services for managing the target hardware, software resources and the implementation of modeled systems based on the supported dataflow MoCs. As discussed, OpenCL offers a programming model consisting of a host and several kernels where the host is a centralized entity that is connected to one or more computing devices and is responsible for the execution of kernels [33].

The framework adopts this idea of host and kernels for the synthesis as shown in Fig. 7. The synthesis method uses a combination of different code generators which generates an OpenCL kernel for each process in the network based on the underlying dataflow MoC of that process. In particular, the generated kernel implements the execution behavior of the process. A single back-end based on OpenCL is developed that provides different specialized code generators for specific dataflow MoCs. Each code generator generates kernel code based on its underlying dataflow MoC. The runtime system systematically employs OpenCL as an integral part of the synthesis and manages the scheduling of processes and their communication based on the dataflow MoC of each process. A scheduler is designed for each dataflow MoC that schedules a process based on the triggering semantics of the underlying MoC. The runtime system is organized in a centralized host and kernels architecture under the OpenCL abstraction. The host accommodates different essential components along with the *Runtime-Manager*. The Runtime-Manager exploits other components of the host and provides different low-level implementations to finally execute the modeled DPNs (kernels) on the target hardware. The back-end comprising of different code generators and the runtime system based on OpenCL are presented in Sect. "Synthesis: The Toolchain".

Modeling: Dataflow Models

The target dataflow process network (DPN) model is based on a limited subset of CAL that is comprised of stateless processes having guarded actions. The main purpose of this section is not to present the formal specification of dataflow models of computation (MoCs) as this has been thoroughly considered in the literature [17, 18]. Instead, the main idea here is to informally illustrate how CAL is used to specify general DPNs and how this general model is restricted to specify different classes of dataflow MoCs. We, therefore, first present the syntax and the informal semantics of the general DPN model based on the used CAL subset and then illustrate the constraints to specify the supported dataflow MoCs.

The General Model of DPN

Recall that a DPN is a set of processes $\mathcal{P} = \{p_0, \ldots, p_{m-1}\}$ with static point-to-point connections via FIFO buffers $\mathcal{F} = \{f_0, \ldots, f_{n-1}\}$. We also assume a total order $\preceq$ on the FIFO buffers so that we can unambiguously switch from sets to tuples of FIFO buffers by simply ordering the corresponding set to a tuple. For this reason, we often ignore the difference between sets and the corresponding tuples. For any tuple $t = (t_0, \ldots, t_\ell)$, we denote its components as $t_i = \mathsf{proj}_i(t)$. Processes of the DPN communicate with each other by consuming data tokens from their input buffers and adding data tokens to the output buffers. Therefore, we define for each process $p \in \mathcal{P}$, the tuple of its input buffers $\mathsf{inBuf}(p)$ and its output buffers $\mathsf{outBuf}(p)$.

In the following, we informally present and elaborate the syntax and the semantics of the general model of a process based on the used subset of CAL.

Syntax

The syntax of a process $p \in \mathcal{P}$ based on the used subset of CAL is illustrated with an abstract example as shown in Listing 2. A process generally consists of a set of input and output buffers and several actions.

```
1  actor ex() <Type> X_1, ..., <Type> X_M ==> <Type>
       Y_1, ..., <Type> Y_N :
2  label_α: action
       X_1 : [x_{1,1}, ..., x_{1,p_1}], ..., X_m : [x_{m,1}, ..., x_{m,p_m}]
       ==>
       Y_1 : [y_{1,1}, ..., y_{1,q_1}], ..., Y_n : [y_{n,1}, ..., y_{n,q_n}]
3    guard γ
4    do
5      y_{1,1} := e_{1,1};
       ⋮
6      ⋮
7      y_{1,q_1} := e_{1,q_1};
       ⋮
8      ⋮
9      y_{n,q_n} := e_{n,q_n};
10   end
   ⋮
11 ⋮
12 label_α: action
       X_1 : [x_{1,1}, ..., x_{1,p_1}], ..., X_m : [x_{m,1}, ..., x_{m,p_m}]
       ==>
       Y_1 : [y_{1,1}, ..., y_{1,q_1}], ..., Y_n : [y_{n,1}, ..., y_{n,q_n}]
13   guard γ
14   do
15     y_{1,1} := e_{1,1};
       ⋮
16     ⋮
17     y_{1,q_1} := e_{1,q_1};
       ⋮
18     ⋮
19     y_{n,q_n} := e_{n,q_n};
20   end
21 endactor
```

Listing 2 Abstract example of a process based on the general CAL DPN model.

Semantics

The abstract example of a process as shown in Listing 2 illustrates the general model based on the used subset of CAL. The head of a process $p \in \mathcal{P}$ specifies the input buffers $\mathsf{inBuf}(p) = (X_1, \ldots, X_M)$ and output buffers $\mathsf{outBuf}(p) = (Y_1, \ldots, Y_N)$, including the type of tokens communicated via the buffer (Line 1). The used CAL subset provides three data types: Boolean, integer and real numbers. The behavior of every process $p \in \mathcal{P}$ is determined by a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. Actions are preceded by action labels which in the general model need not to be unique, i.e., the same label can be used for more than one action (Lines 2 and 12). The head of an action $\alpha \in \mathsf{actions}(p)$ specifies for the input buffers in $\mathsf{inBuf}(p)$ the number of data tokens to be read (Line 2). It may or may not specify all input buffers in $\mathsf{inBuf}(p)$. If the action is fired, these data tokens are consumed from the heads of input buffers and are assigned to the variables $x_{i,j}$ such that $x_{i,1}$ is the head of the input buffer X_i. Analogously, the action interface determines for the output buffers in $\mathsf{outBuf}(p)$ the number of data tokens to be written. Thereby, the values $y_{i,1}, \ldots, y_{i,q_i}$ are added in this order to the tail of output buffer Y_i. The body of the action is, therefore, a sequence of statements that compute values based on expressions $e_{i,1}, \ldots, e_{i,q_i}$ and assign them to output variables $y_{i,1}, \ldots, y_{i,q_i}$ (Lines 5–9). An expression may compose of variables, values, and both arithmetic and Boolean expressions. Since only a subset of $\mathsf{inBuf}(p)$ may be used by an action $\alpha \in \mathsf{actions}(p)$, we also define $\mathsf{inAct}(\alpha) \subseteq \mathsf{inBuf}(p)$ as the subset of input buffers used by that action. Similarly, we define $\mathsf{outAct}(\alpha) \subseteq \mathsf{outBuf}(p)$ as the subset of output buffers used by the action. For an action $\alpha \in \mathsf{actions}(p)$ that requires that input tokens have particular values, an additional condition can be specified using a guard (Line 3) which is a predicate on the tokens of (some prefixes of) the input buffers in $\mathsf{inAct}(\alpha)$. Since only a subset of $\mathsf{inAct}(\alpha)$ may be used by a guard, we also define $\mathsf{inGrd}(\alpha) \subseteq \mathsf{inAct}(\alpha)$ as the subset of input buffers whose values are considered by the guard γ_α of action α.

For the semantics, we consider a domain $\mathcal{D}$ of values that may be the union of integers, booleans and real numbers. We denote the set of finite sequences on $\mathcal{D}$ as $\mathcal{D}^*$ and the set of infinite sequences on $\mathcal{D}$ as $\mathcal{D}^\omega$, and the union of both as $\mathcal{D}^\infty$, i.e., $\mathcal{D}^\infty := \mathcal{D}^* \cup \mathcal{D}^\omega$. For sequences $\sigma_1, \sigma_2 \in \mathcal{D}^\infty$, we introduce the prefix ordering $\sigma_1 \sqsubseteq \sigma_2 :\Leftrightarrow \exists \sigma_3 \in \mathcal{D}^\infty.\ \sigma_2 = \sigma_1 \cdot \sigma_3$ where $\sigma_1 \cdot \sigma_3$ means the concatenation of the sequences σ_1 and σ_3 which demands that $\sigma_1 \in \mathcal{D}^*$. The prefix ordering on sequences $\sigma_1, \sigma_2 \in \mathcal{D}^\infty$ is lifted to tuples of sequences $\sigma_1 = (\sigma_{1,0}, \ldots, \sigma_{1,\ell})$ and $\sigma_2 = (\sigma_{2,0}, \ldots, \sigma_{2,\ell})$ in that we demand $\sigma_{1,i} \sqsubseteq \sigma_{2,i}$ for all $i \in \{0, \ldots, \ell\}$.

Each process $p \in \mathcal{P}$ defines a function that maps the consumed input tokens to produced output tokens. This function is determined by a set of actions $\mathsf{actions}(p)$ of the process p where the semantics of each action $\alpha \in \mathsf{actions}(p)$ is a function of type $(\mathcal{D}^*)^m \to (\mathcal{D}^*)^n$ with the following meaning: The action consumes tokens from m input buffers and produces tokens to n output buffers, thus, $m := |\mathsf{inAct}(\alpha)|$ and $n := |\mathsf{outAct}(\alpha)|$.

Any action $\alpha \in \mathsf{actions}(p)$ as shown in Listing 2 is enabled iff the following conditions are all satisfied:

- each input buffer $X_i \in \mathsf{inAct}(\alpha)$ has enough tokens, i.e., X_i must have at least p_i many tokens
- each output buffer $Y_i \in \mathsf{outAct}(\alpha)$ has enough space, i.e., Y_i must have at least space for q_i many tokens
- the guard condition γ which is a condition on the input tokens $x_{i,j}$ in $\mathsf{inGrd}(\alpha)$ is satisfied

```
1.  actor nondet-merge() int X1, int X2 ==> int Y1
2.  act1: action X1: [x1] ==> Y1: [y1]
3.      guard true
4.      do
5.        y1:= x1;
6.      end
7.  act2: action X2:[x2] ==> Y1: [y1]
8.      guard true
9.      do
10.       y1:= x2;
11.     end
12. end
```

Fig. 8 A simple example of a non-deterministic process in CAL. The output produced on **Y1** depends on the arrival time of tokens on the inputs **X1** and **X2**

The general model of DPN does not impose further restrictions and, therefore, actions consisting of common inputs and/or outputs may be enabled in the same execution, as depicted in Listing 2. As a result, this gives rise to read and write conflicts in buffers, ultimately ending up in non-deterministic behaviors. A read conflict means that two actions are enabled in an execution that read a token from the same input. Whereas, a write conflict means that two actions are enabled in an execution that write a token to the same output. A simple example of a non-deterministic process is illustrated in Fig. 8. It consists of two actions *act1* and *act2* that consume tokens from different inputs *X1* and *X2*, respectively, and produce tokens to the common output *Y1*. Depending on the availability of tokens on the inputs, both actions may be enabled in the same execution, and, therefore, may give rise to write conflict in *Y1*. Hence, the output produced on *Y1* depends on the arrival time of tokens on the inputs and, therefore, exhibits a non-deterministic behavior.

We, therefore, demand and use the general CAL DPN model with specific constraints and restrictions to specify the precise dataflow MoCs.

Static Dataflow Model

The static dataflow (SDF) [26] MoC allows one to model static (synchronous) behaviors. It is a more restricted DPN class such that the decision on whether to consume and produce tokens in each execution can be made statically at compile-time. Each execution of a process consumes and produces a fixed number of tokens. The number of tokens consumed or produced on each buffer must be independent of the value as well as the arrival time of data. A process in SDF becomes enabled if and only if all its inputs have required tokens and all its outputs have required space. An enabled process may fire, and once fired, consumes the statically specified number of tokens from its inputs and produces the statically specified number of tokens to its outputs.

We demand and assume certain restrictions on the general DPN model to represent the SDF MoC. In the following, we present an abstract example of a static process based on the SDF MoC and informally illustrate its semantics.

Syntax

The syntax of a static process in SDF is illustrated with an abstract example as shown in Listing 3.

```
1  actor SDF() <Type> X_1,...,<Type> X_m ==> <Type>
       Y_1,...,<Type> Y_n:
2  label_{α_1}: action
       X_1 : [x_{1,1}, ..., x_{1,p_1}], ..., X_m : [x_{m,1}, ..., x_{m,p_m}]
       ==>
       Y_1 : [y_{1,1}, ..., y_{1,q_1}], ..., Y_n : [y_{n,1}, ..., y_{n,q_n}]
3    guard γ_{α_1}
4    do
5      y_{1,1} := e_{1_{1,1}};
6      ⋮
7      y_{1,q_1} := e_{1_{1,q_1}};
8      ⋮
9      y_{n,q_n} := e_{1_{n,q_n}};
10   end
11 ⋮
12 label_{α_h}: action
       X_1 : [x_{1,1}, ..., x_{1,p_1}], ..., X_m : [x_{m,1}, ..., x_{m,p_m}]
       ==>
       Y_1 : [y_{1,1}, ..., y_{1,q_1}], ..., Y_n : [y_{n,1}, ..., y_{n,q_n}]
13   guard γ_{α_h}
14   do
15     y_{1,1} := e_{h_{1,1}};
16     ⋮
17     y_{1,q_1} := e_{h_{1,q_1}};
18     ⋮
19     y_{n,q_n} := e_{h_{n,q_n}};
20   end
21 endactor
```

Listing 3 Abstract example of a process in SDF.

Semantics

A process $p \in \mathcal{P}$ in SDF consists of a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. The action labels need to be unique, i.e., the same label must not be used for more than one action (Lines 2 and 12). For each action α_i, we define its guard γ_{α_i}. Each action $\alpha_i \in \mathsf{actions}(p)$ specifies for all input buffers $\mathsf{inBuf}(p)$ and all output buffers $\mathsf{outBuf}(p)$ the number of tokens to be read and written, respectively. Thus, the input and output buffers are always same across all actions i.e., $\mathsf{inAct}(\alpha_1) = \ldots = \mathsf{inAct}(\alpha_h)$ and $\mathsf{outAct}(\alpha_1) = \ldots = \mathsf{outAct}(\alpha_h)$. Second, the number of tokens to be consumed and the number of tokens to be produced are always same for the same input and output buffers, respectively, across all actions. This restricts the execution of processes to fixed consumption and production rates.

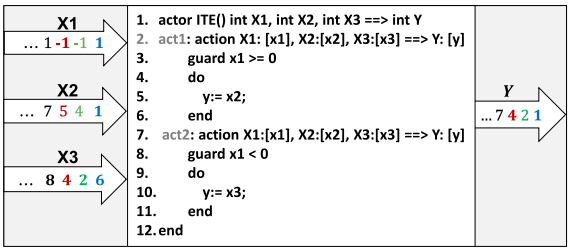

Fig. 9 The static if-then-else (ITE) node: a simple example of a static process in SDF. An example behavior is illustrated with a set of input values and the computed output values as shown inside arrows

Regardless of which action is executed, the same number of tokens are consumed and produced in the same buffers in each firing of a process. Moreover, we demand that the guard conditions should always be mutually exclusive across actions. This ensures that for each execution of a process, the actions will never compete for an execution. Hence, in each execution of a process only a specific action is fired whose guard is enabled.

Execution of Actions

Each time a process $p \in \mathcal{P}$ is triggered for an execution, a particular action is executed, mainly dependent on which guard is enabled. The guards of actions $\mathsf{actions}(p)$ are always evaluated sequentially in the same order of their actions definitions. Since all actions in a process have same input buffers with same consumption rates, hence for any action $\alpha_i \in \mathsf{actions}(p)$, the specified fixed number of tokens are first consumed from all input buffers $\mathsf{inAct}(\alpha_i) = \mathsf{inBuf}(p)$. Finally, the enabled action is fired whose guard is true. Upon firing, the defined computations are performed and the specified fixed number of tokens are produced to all output buffers $\mathsf{outAct}(\alpha_i) = \mathsf{outBuf}(p)$.

Triggering Processes for Execution

Each process $p \in \mathcal{P}$ in SDF is triggered for an execution if and only if all input buffers $\mathsf{inAct}(\alpha_i)$ of an action $\alpha_i \in \mathsf{actions}(p)$ have enough input tokens and all output buffers $\mathsf{outAct}(\alpha_i)$ of that action have enough space. The process shown in Listing 3 is triggered for an execution iff for any action α_i, each input buffer $X_j \in \mathsf{inAct}(\alpha_i)$ has at least p_j many tokens and each output buffer $Y_j \in \mathsf{outAct}(\alpha_i)$ has at least space for q_j many tokens.

SDF Process Example

A simple example of the static if-then-else (ITE) operation is illustrated in Fig. 9. In each execution, the ITE process consumes a token each from all three inputs and produces a token to its only output. It consists of two actions i.e., *act1* and *act2*, having same inputs *X1*, *X2* and *X3*, and the same output *Y* (Lines 2 and 7). Both actions use the input *X1* for the guard with mutually exclusive guard conditions (Lines 3 and 8). In each execution, depending on which guard is enabled, either *act1* or *act2* fires for an execution. ITE is only triggered for an execution if there is a token available in all three inputs *X1*, *X2* and *X3* and if there is space available for a token to be produced at the output *Y*. The tokens are denoted by small letters *x1*, *x2*, *x3* and *y*.

Kahn Process Networks Model

Kahn process networks (KPNs) [21] are dynamic DPNs where processes can consume and produce different number of tokens in every firing depending on the history of the consumed tokens and also on the tokens to be consumed. KPNs exhibit latency-insensitive deterministic behaviors that do not depend on the timing or the execution order of the processes. The KPN MoC is typically specified with the following restrictions and properties:

- processes are not allowed to test input buffers for the existence of tokens
- reading from input buffers is blocking, and writing to output buffers is non-blocking
- processes must implement deterministic sequential functions
- processes do not need all of their inputs to get triggered for execution

Based on these restrictions/properties, it can be implied that a process in KPN can be any sequential program where the firing rules can be tested sequentially in a predefined order in each execution using blocking reads [27]. This reflects the ability to uniquely consume the inputs in each firing without timing information provided by the input signals. A KPN process is only triggered for execution if the exact information on inputs required to produce the output is available. A process, therefore, becomes enabled if the required values on inputs are available to perform the computation and produce the output. A process once enabled, may fire, and once fired, it may consume different number of tokens from different inputs based on the history of the consumed tokens. The KPN MoC can capture both static as well as sequential behaviors. Since buffers with unbounded capacity cannot be realized in real implementations, the used KPN model only supports blocking write. However, since the starting point of this work is the desynchronized models, desynchronization preserves properties like deadlock-freedom and bounded memory usage that are otherwise difficult to ensure in KPNs [2].

Next, we present an abstract example of a sequential process based on the KPN MoC and illustrate its semantics.

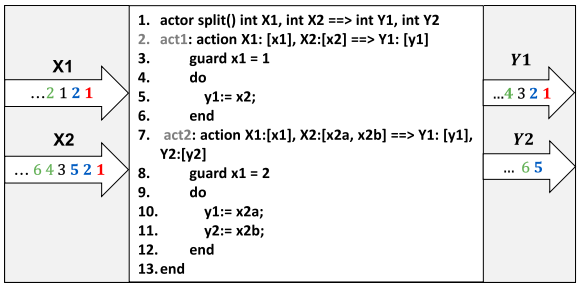

Fig. 10 The split node: a simple example of a sequential process in KPN. An example behavior is illustrated with a set of input values and the computed output values as shown inside arrows

Syntax

The syntax of a sequential process in KPN is illustrated with an abstract example as shown in Listing 4.

```
1  actor KPN() <Type> X_1,...,<Type> X_M ==> <Type> Y_1,...,<Type> Y_N :
2  label_α1 : action X_1 : [x_{1,1},...,x_{1,p_1}],...,X_m : [x_{m,1},...,x_{m,p_m}]
       ==> Y_1 : [y_{1,1},...,y_{1,q_1}],...,Y_n : [y_{n,1},...,y_{n,q_n}]
3    guard γ_α1
4    do
5      y_{1,1} := e_{1_{1,1}};
       ⋮
6      y_{1,q_1} := e_{1_{1,q_1}};
       ⋮
7      y_{n,q_n} := e_{1_{n,q_n}};
8    end
9  ⋮
10 label_α_h : action X_1 : [x_{1,1},...,x_{1,f_1}],...,X_u : [x_{u,1},...,x_{u,f_u}]
       ==> Y_1 : [y_{1,1},...,y_{1,g_1}],...,Y_v : [y_{v,1},...,y_{v,g_v}]
11   guard γ_α_h
12   do
13     y_{1,1} := e_{h_{1,1}};
       ⋮
14     y_{1,g_1} := e_{h_{1,g_1}};
       ⋮
15     y_{v,g_v} := e_{h_{v,g_v}};
16   end
17 endactor
```

Listing 4 Abstract example of a process in KPN.

Semantics

A process $p \in \mathcal{P}$ in KPN consists of a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. The action labels need to be unique, i.e., the same label must not be used for more than one action (Lines 2 and 12). For each action α_i, we define its guard γ_{α_i}. Each action $\alpha_i \in \mathsf{actions}(p)$ specifies for the input buffers $\mathsf{inAct}(\alpha_i) \subseteq \mathsf{inBuf}(p)$ and the output buffers $\mathsf{outAct}(\alpha_i) \subseteq \mathsf{outBuf}(p)$ the number of tokens to be read and written, respectively. In general, the input and output buffers can be different across different actions. However, since processes in KPN consist of sequential functions, we demand that all actions in a process must have at least one common input. This implies that $\mathsf{inAct}(\alpha_1) \cap \ldots \cap \mathsf{inAct}(\alpha_h) \neq \{\}$. Moreover, we demand that the guard conditions are always mutually exclusive across actions. This ensures that for each execution of a process, the actions will never compete for an execution. Hence, in each firing of a process only a specific action is executed mainly dependent on which guard is enabled. Second, this enables the execution of processes with dynamic consumption rates and dynamic production rates, mainly dependent on which guards are enabled on each execution.

Evaluation and Execution of Actions

As discussed, the KPN MoC does not allow processes to test input buffers for the existence of tokens. A process is only triggered for execution if the exact information on inputs required to execute an action is available. Therefore, each time a process $p \in \mathcal{P}$ is triggered for an execution, a particular action $\alpha_i \in \mathsf{actions}(p)$ is executed whose guard γ_{α_i} is enabled. The enabled action α_i, once fires, consumes a finite number of tokens from the input buffers $\mathsf{inAct}(\alpha_i)$ and produces a finite number of tokens to the output buffers $\mathsf{outAct}(\alpha_i)$ as specified for that action.

Triggering Processes for Execution

Since processes in KPNs consist of sequential programs, the availability of tokens on the inputs, the availability of space on the outputs, and the guards can be evaluated sequentially in a predefined order of their actions definitions. Each process $p \in \mathcal{P}$ is triggered for an execution if there exists one particular action $\alpha_i \in \mathsf{actions}(p)$ having: enough input tokens in $\mathsf{inAct}(\alpha_i)$, required values on the guarded inputs $\mathsf{inGrd}(\alpha_i)$, and enough space in $\mathsf{outAct}(\alpha_i)$. For instance, the process shown in Listing 4 is triggered for an execution when for a particular action (say α_h), each input buffer $X_j \in \mathsf{inAct}(\alpha_h)$ has at least f_j many tokens, each output buffer $Y_j \in \mathsf{outAct}(\alpha_h)$ has at least space for g_j many tokens, and the guard γ_{α_h} is true. In case if one of the inputs does not have enough tokens, the process is blocked (i.e., the blocking behavior of KPN) until sufficient tokens are available on that input.

KPN Process Example

A simple example of a sequential process, namely the *split* node is illustrated in Fig. 10 that splits a single input channel to a number of individual output channels. The split node consists of two actions *act1* and *act2* that depending on the value of a token at the input *X1* splits the tokens from the

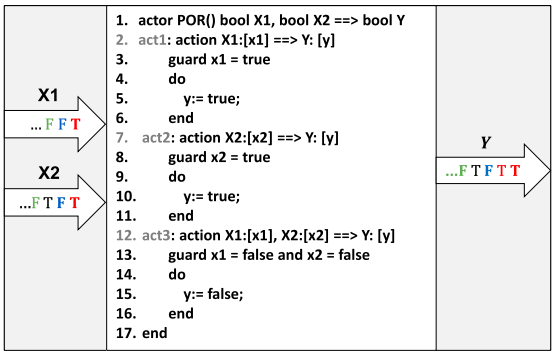

Fig. 11 The parallel OR node: a simple example of a well-behaved parallel process in DDF. An example behavior is illustrated with a set of input values and the computed output values as shown inside arrows

input *X2* to outputs *Y1* and *Y2*. The guards are composed of mutually exclusive conditions (Lines 3 and 8). Both actions declare the input *X2* with different consumption rates (Lines 2 and 7). The action *act2* has an additional output *Y2*. In each execution, depending on which guard is enabled, either *act1* or *act2* fires for an execution. In the case where *act1* fires, a single token each is consumed from *X1* and *X2*, and a single token is produced to *Y1* (Line 5). On the contrary, when *act2* fires, a single token is consumed from *X1*, two tokens are consumed from *X2* and a token each is produced to *Y1* and *Y2* (Lines 10–11). Hence, in each execution, a different number of tokens can be consumed from the input *X2* and a different number of tokens can be produced to outputs *Y1* and *Y2*. The split node is only triggered for an execution if there exists one action i.e., either *act1* or *act2*, having required number of tokens in *X1* and *X2*, required space in outputs *Y1* and *Y2*, and required values on the input *X1*.

Dynamic Dataflow Model

The dynamic dataflow (DDF) also sometimes referred to as the non-determinate dataflow is a dynamic DPN class that allows one to model dynamic and asynchronous processes. It offers a more generalized data dependent and asynchronous execution semantics than the KPN MoC. In particular, the DDF MoC allows one to model processes with parallel programs consisting of concurrent and independent computations where more than one action can be executed in each firing. This generalization results in higher expressiveness and flexibility, however, may lead to non-deterministic behaviors, e.g., a non-determinate merge [27]. We use a variant of the DDF MoC that only supports concurrent and independent actions with specific restrictions. It can be used to model well-behaved parallel nodes that exhibit deterministic behaviors, e.g., the parallel OR (POR) node as illustrated in Fig. 11 (discussed in the next subsection).

```
1  actor DDF() <Type> X_1, ..., <Type> X_m, ..., <Type
       > X_M ==> <Type> Y_1, ..., <Type> Y_n, ..., <
       Type> Y_N :
2  label_α1 : action
       X_1 : [x_{1,1}, ..., x_{1,p_1}], ..., X_m : [x_{m,1}, ..., x_{m,p_m}]
       ==>
       Y_1 : [y_{1,1}, ..., y_{1,q_1}], ..., Y_n : [y_{n,1}, ..., y_{n,q_n}]
3    guard γ_α1
4    do
5      y_{1,1} := e_{1_{1,1}};
6      ⋮
7      y_{1,q_1} := e_{1_{1,q_1}};
8      ⋮
9      y_{n,q_n} := e_{1_{n,q_n}};
10   end
11 ⋮
12 label_αk : action X_{m+1} : [x_{m+1,1}, ..., x_{1,p_{m+1}}], ...,
       X_M : [x_{M,1}, ..., x_{M,p_M}] ==>
       Y_{n+1} : [y_{n+1,1}, ..., y_{n+1,q_{n+1}}], ...,
       Y_N : [y_{N,1}, ..., y_{N,q_N}]
13   guard γ_αk
14   do
15     y_{n+1,1} := e_{k_{n+1,1}};
16     ⋮
17     y_{n+1,q_{n+1}} := e_{k_{n+1,q_{n+1}}};
18     ⋮
19     y_{N,q_N} := e_{k_{N,q_N}};
20   end
21 ⋮
22 label_αh : action X_1 : [x_{1,1}, ..., x_{1,f_1}], ...,
       X_m : [x_{m,1}, ..., x_{m,f_m}] ...,
       X_M : [x_{M,1}, ..., x_{M,g_M}] ==>
       Y_1 : [y_{1,1}, ..., y_{1,q_1}], ..., Y_n : [y_{n,1}, ..., y_{n,q_n}]
       ... Y_N : [y_{N,1}, ..., y_{N,q_N}]
23   guard γ_αh
24   do
25     y_{1,1} := e_{h_{1,1}};
26     ⋮
27     y_{1,q_1} := e_{h_{1,q_1}};
28     ⋮
29     y_{N,q_N} := e_{h_{N,q_N}};
30   end
31 endactor
```

Listing 5 Abstract example of a process in DDF.

The considered variant of DDF MoC offers a more flexible semantics where each process becomes enabled for an execution if only one of its inputs has required tokens and only one of its outputs has required space. The decision on whether to consume/produce tokens and to execute each action of an enabled process is made dynamically at runtime when that particular process is triggered for an execution. A process once enabled, may fire, and once fired, it may trigger multiple actions for execution. A process can produce and consume different number of tokens in every firing. The considered variant of DDF MoC can capture static, sequential and well-behaved parallel processes.

In the following, we present an abstract example of a well-behaved parallel process based on the DDF MoC and informally illustrate its semantics.

Syntax

The syntax of a well-behaved parallel process in DDF is illustrated with an abstract example as shown in Listing 5.

Semantics

A process $p \in \mathcal{P}$ in DDF consists of a set of actions $\mathsf{actions}(p) = \{\alpha_1, \ldots, \alpha_h\}$. The action labels need to be unique, i.e., the same label must not be used for more than one action (Lines 2, 12 and 22). For each action α_i, we define its guard γ_{α_i}. Different actions in $\mathsf{actions}(p)$ may specify completely different input and output buffers e.g., $\mathsf{inAct}(\alpha_1) \cap \mathsf{inAct}(\alpha_k) = \{\}$ and $\mathsf{outAct}(\alpha_1) \cap \mathsf{outAct}(\alpha_k) = \{\}$. This enables the modeling of processes with independent actions consisting of completely different inputs and outputs. Moreover, different actions in $\mathsf{actions}(p)$ may also specify common input and output buffers e.g., $\mathsf{inAct}(\alpha_1) \cap \mathsf{inAct}(\alpha_h) \neq \{\}$ and $\mathsf{outAct}(\alpha_k) \cap \mathsf{outAct}(\alpha_h) \neq \{\}$. The associated number of token variables (i.e., token consumption rates) of common input buffers can be different across actions. Multiple actions may fire in the same execution of a process. However, we demand that these firings must be consistent and do not give rise to non-deterministic behaviors. In particular, we demand that the guard conditions of actions with common input buffers are always mutually exclusive. Hence, in each firing of a process only a specific action from all actions having at least one common input buffer is executed whose guard is enabled.

This ensures that for each execution of a process, the actions with common input buffers will never compete for an execution for any set of tokens. Moreover, for actions with at least one common output buffer, we demand that each action upon firing produces the same sequence of tokens at the common output buffer. Hence, the firing of actions with common output buffers do not lead to different output streams. Altogether, these restrictions enable the execution of processes consisting of well-behaved parallel programs with dynamic consumption rates and dynamic production rates, mainly dependent on which guards are enabled on each execution. An example of a well-behaved parallel node exhibiting such a behavior is illustrated in Fig. 11.

Evaluation and Execution of Actions

Each action $\alpha_i \in \mathsf{actions}(p)$ of a process $p \in \mathcal{P}$ is evaluated for an execution dynamically when that particular process is triggered for an execution. Each action α_i fires for an execution iff: there are enough tokens available in $\mathsf{inAct}(\alpha_i)$, enough space available in $\mathsf{outAct}(\alpha_i)$, and the required values on the guarded inputs $\mathsf{inGrd}(\alpha_i)$ are available, i.e., the guard γ_{α_i} is true. When the action α_i fires, it consumes a finite number of tokens from $\mathsf{inAct}(\alpha_i)$ and produces a finite number of tokens to $\mathsf{outAct}(\alpha_i)$.

Since the actions are evaluated dynamically only after a particular process is triggered for an execution, there may be a case when for an action α_i, although the guard γ_{α_i} is true, however, either at least one of the input buffers in $\mathsf{inAct}(\alpha_i)$ does not have enough tokens, or at least one of the output buffers in $\mathsf{outAct}(\alpha_i)$ does not have enough space. In this case, neither the input tokens are consumed from $\mathsf{inAct}(\alpha_i)$, nor the output tokens are produced to $\mathsf{outAct}(\alpha_i)$. Instead, the tokens are preserved in their respective input buffers.

Triggering Processes for Execution

As the decision to execute each action of a process and consume/produce data tokens is made dynamically at runtime, when that process is triggered for an execution. Therefore, each process $p \in \mathcal{P}$ is triggered for an execution if there exists at least one input buffer $X_j \in \mathsf{inAct}(\alpha_i)$ having enough input tokens and there exists at least one output buffer $Y_j \in \mathsf{inAct}(\alpha_i)$ having enough space. For instance, the process shown in Listing 5 is triggered for an execution when for a particular action (say α_1), any input buffer $X_j \in \mathsf{inAct}(\alpha_1)$ has at least p_j many tokens and any output buffer $Y_j \in \mathsf{outAct}(\alpha_1)$ has at least space for q_j many tokens.

DDF Process Example

A simple example of the parallel OR (POR) node is illustrated in Fig. 11 that performs the logical OR operation on two Boolean inputs. The POR node consists of three actions *act1*, *act2* and *act3* that depending on the values of tokens in either or both of the inputs *X1* and *X2* produces tokens in the only output *Y*. The actions *act1* and *act3* share a common input *X1* (Lines 2 and 12). The actions *act2* and *act3* share a common input *X2* (Lines 7 and 12). All actions share the same output *Y* (Lines 2, 7 and 12). In each execution, depending on which guards are enabled, either one or both of the actions *act1*, *act2* can be fired. In case if both the actions are enabled, a token each is produced to the output *Y* by both actions (Lines 5 and 10). Since they share a common output, hence, the same sequence of tokens is produced at the output *Y* by both actions. In contrast, if the guard is true for *act3*, the actions *act1*, *act2* become disabled where only the action *act3* is fired. Therefore, the actions with common inputs never compete for firing in any execution of the process. The POR node is only triggered for an execution if there is a token available in at least one of the inputs i.e., either *X1* or *X2*, and there is a space available for one token to be produced in the only output *Y*.

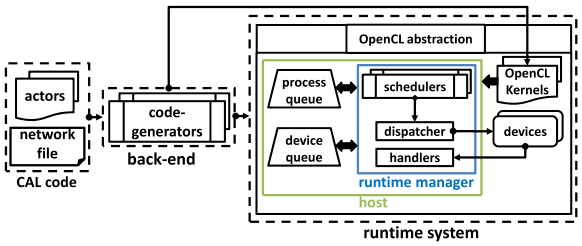

Fig. 12 The proposed synthesis tool chain is composed of two main parts: the *back-end* features the specialized code generators for the supported dataflow MoCs. The *runtime system* organized in a host and kernels program model schedules and maps the modeled system on the target hardware

Synthesis: The Toolchain

In this section, we present the synthesis tool chain which is depicted in Fig. 12. It offers a set of tools that work together to finally implement the CAL DPNs on commercial off-the-shelf (COTS) target hardware. The tool chain is composed of two main parts: a special *back-end* comprising of specialized code generators for particular dataflow MoCs and the *runtime system*. Each code generator generates an OpenCL kernel for each process based on the underlying dataflow MoC. Second, the runtime system is organized in a centralized host and kernels program model, built under the OpenCL abstraction. The host features different components including the *Runtime-Manager* that schedule and deploy processes (generated kernels) on the target hardware.

Back-end

The back-end is designed to work in two different modes, namely the *manual* mode and the *auto* mode. In the manual mode, the back-end targets homogeneous implementations based on a specific user given dataflow MoC. In the auto mode, the back-end automatically classifies the processes into three categories mainly according to their kinds of behaviors that determine the dataflow MoCs. This classification of different kinds of behaviors involves: the static ones based on the static dataflow (SDF) MoC, the sequential ones based on the Kahn process network (KPN) MoC, and the parallel ones based on a variant of the dynamic dataflow (DDF) MoC. As a result, the back-end provides three different specialized code generators for particular dataflow MoCs: one for the SDF MoC, second for the KPN MoC, and finally for the used DDF MoC. We identify the kinds of behaviors of processes during desynchronization based on a succinct formalization of input/output (I/O) firing rules of synchronous components [38, 39]. The identified dataflow MoC of each process is divulged to the synthesis phase through the network description file.

In the following, we present the specialized code generators based on the supported dataflow MoCs.

Code Generators: Kernel Code Generation

Each code generator is determined by the underlying semantics of the used dataflow MoC. It, therefore, generates an OpenCL kernel for each process based on the underlying dataflow MoC. This section presents the schemes employed for generating OpenCL kernel code based on all the supported dataflow MoCs. Moreover, we also illustrate the generated kernel code for each supported dataflow MoC based on the CAL models presented in Figs. 9, 10 and 11.

Each code generator generates kernel code in two segments: First, the OpenCL specific code is targeted which involves the generation of the kernel header, the declaration of used inputs and outputs, and most importantly, the generation of generic kernel code that enables the host to dispatch multiple executions of the kernel on the device. This code segment is more or less same for all used dataflow MoCs. The second segment targets the code generation based on the underlying semantics of the used dataflow MoC. We mainly present the schemes designed for generating code for the latter segment. The former segment is explained in detail with the generated kernel code examples.

Algorithm 1: Pseudo code for SDF kernel code generation of a process p.

```
1 ConsumeTokens(inBuf(p));
2 foreach action α in p do
3 |   EvaluateGuard(γ_α);
4 |   if GuardValid(γ_α) then
5 |   |   PerformComputations(α);
6 |   |   ProduceTokens(outAct(α));
7 |   end
8 end
```

SDF Code Generator

In SDF, it is statically determined that each firing of a process consumes/produces fixed number of tokens. A process in SDF is, therefore, simply scheduled by the host for execution if there is enough data available in all inputs and if there is enough space available in all outputs. The guards of actions are evaluated within kernels on the device side. The SDF code generation is relatively straightforward and is illustrated by the pseudo code given in Algorithm 1.

For a static process, the proposed scheme works as follows: First, the code is generated to consume tokens from all input buffers of the process (Line 1). The algorithm then iterates through the set of modeled actions in the order of their definitions (Line 2) where for each action, it proceeds as follows: First, the code is generated to evaluate the guard (Line 3). Next, the code is generated for the case if the guard

is fulfilled (Lines 4–7). To this end, the code to perform the modeled computations is generated (Line 5), and then to produce the final output (Line 6). The generated kernel for a static process *ITE* as shown in Fig. 9 is listed in Listing 6.

```
1  __kernel void ITE ( __global fifo_t* X1 ,
       __global fifo_t* X2 , __global fifo_t* X3,
        __global fifo_t* Y, int blockSize)
2  {
3    /* Declarations for All Inputs and Outputs */
4    __private int seq_X1[1];
5    __private int seq_X2[1];
6    __private int seq_X3[1];
7    __private int seq_Y[1];
8    int* x1_act1 = &seq_X1[0];
9    int* x1_act2 = &seq_X1[0];
10   int* x2_act1 = &seq_X2[0];
11   int* x2_act2 = &seq_X2[0];
12   int* x3_act1 = &seq_X3[0];
13   int* x3_act2 = &seq_X3[0];
14   int* y_act1 = &seq_Y[0];
15   int* y_act2 = &seq_Y[0];
16   /* Generic Kernel Code */
17   int gid = get_global_id(0) * blockSize;
18   for (int x = 0; x < blockSize; x++) {
19     /* SDF Specific Kernel Code */
20     fifoRead(X1, seq_X1, 1, gid);
21     fifoRead(X2, seq_X2, 1, gid);
22     fifoRead(X3, seq_X3, 1, gid);
23     if(*x1_act1 >= 0 ) {
24       *y_act1 = *x2_act1;
25       fifoWrite(Y, seq_Y, 1, gid);
26     }
27     else if(*x1_act2 < 0 ) {
28       *y_act2 = *x3_act2;
29       fifoWrite(Y, seq_Y, 1, gid);
30     }
31   }
32 }
```

Listing 6 Generated kernel for *ITE* as illustrated in Fig. 9.

OpenCL specific code segment. This segment (Lines 1–18) is mainly composed of the following parts: The generated kernel header uses the name of the process and defines the argument list (Line 1). The argument list mainly describes the OpenCL memory objects for the input and output FIFO buffers of the process. These memory objects are used by the host for data communication to and from the kernels (device side). These objects are defined with the *global* address space name that allocates them in the global memory shared between the host and devices. For better memory performance, the kernel instances (work-items) do not directly perform operations on the slower global memory. Instead, an array each is declared for the sequences of input/output tokens with the *private* address space name (Lines 4–15), which refers to a faster memory region only visible to individual instances. In each execution of a SDF process, only a particular action is executed, which depends on the enabled guard. To avoid unnecessary duplication, an array is declared for each input/output based on the statically determined consumption/production rate (Lines 4–7). Second, the individual input/output token variables are declared and pointed to their respective sequences i.e., arrays (Lines 8–15).

Moreover, the code generator generates generic kernel code (Lines 17–18) to enable the centralized host to dispatch multiple execution (instances) of kernels on the device at a time. The generic code involve two main components: First, the OpenCL function *get_global_id(0)* provides the unique global ID for the particular kernel instance or thread based on the number of instances dispatched by the host to execute the kernel. These dispatched instances are ideally executed in parallel on the device where each instance operates on data based on its own unique ID. We further introduced an additional parameter, namely *blockSize* (Lines 17–18) that allows us to manage the amount of workload associated with each instance. *blockSize* coalesces multiple instances in a single instance and executes them sequentially inside a loop (Line 18). Consequently, increasing *blockSize* implies increasing the workload per instance and hence decreasing the total number of parallel instances. The combination of the unique global ID and *blockSize* can be effectively used by the host to fine tune the amount of data level parallelism according to the available resources, and most importantly, based on the kinds of behaviors of processes. For example, if the target device is a CPU offering only a few cores, a larger *blockSize* usually achieves better performance [34]. Second, if a process exhibits a dynamic behavior, multiple instances can not be executed in parallel and, therefore, a *blockSize* equal to the number of instances can be used by the host.

SDF MoC specific code segment. This code segment (Lines 19–31) is generated based on the scheme presented in Algorithm 1. First, the tokens are consumed from all inputs of the process (Line 20–22). Finally, a particular action (i.e., either *act1* or *act2*) is executed based on the activation of guard. To this end, if the guard is true for *act1* (Line 23), a token consumed from the input *X2* is written to the output *Y* (Lines 24–25). On the other hand, if the guard is true for *act2* (Line 27), a token consumed from the input *X3* is written to the output *Y* (Lines 28–29).

Algorithm 2: Pseudo code for KPN kernel code generation of a process p.

```
1 foreach action α in p do
2 |   if EvaluatedGuardValid(α) then
3 |   |   ConsumeTokens(inAct(α)) ;
4 |   |   PerformComputations(α);
5 |   |   ProduceTokens(outAct(α)) ;
6 |   end
7 end
```

KPN Code Generator

The KPN MoC supports static as well as sequential behaviors. Since processes in KPNs are sequential, their firing rules (including guards) can be evaluated sequentially in a predefined order. In particular, a process is only triggered by the host for execution if it is already known that the guard of one of the actions is evaluated to true. In contrast to the SDF MoC, the guards are, therefore, evaluated at the time of scheduling on the host side. This essentially simplifies the kernel code generation, however, on the other hand, relatively complicates the scheduling. The code generation based on the underlying semantics of the KPN MoC is illustrated by the pseudo code given in Algorithm 2.

For a process in KPN, the proposed algorithm works as follows: It iterates through the set of modeled actions in the order of their definitions (Line 1) where for each action, it proceeds as follows: First, the code is generated that checks if the already evaluated guard is valid for the action (Line 2). Next, the code is generated for the case if the guard is valid (Lines 3–5). To this end, the code is generated to consume tokens from all inputs of the action (Line 3). Second, the generated code for the modeled computations is inserted (Line 4), prior to generating the code for writing the computed results on the outputs (Line 5). In contrast to the SDF MoC, where data is consumed from all inputs of the process in each execution, the KPN MoC only consumes data from the inputs of an enabled action.

The generated kernel for a sequential process *split* as shown in Fig. 10 is listed in Listing 7.

OpenCL specific code segment. As discussed, this code segment is largely same for all supported dataflow MoCs. In contrast to the SDF MoC, where the guards are evaluated within kernels at the device side, in the case of KPN, the guards are evaluated at the host side typically at the time of scheduling. The host provides the information regarding the evaluated guards to the kernel using a data structure *evaluatedGuard* through the argument list (Line 1). In particular, each element of *evaluatedGuard* holds the identifier for an action whose guard is valid for a particular instance. In each execution of a process, only a particular action is executed, which depends on the enabled guard. To avoid unnecessary duplication, an array is declared for each input/output with the highest consumption/production rate of all actions. For instance, an array is declared for the input *X*2 with the highest consumption rate of both actions (Line 5). The remaining code of this segment is exactly the same as explained for the SDF MoC.

KPN MoC specific code segment. This code segment (Lines 19–34) is generated based on the scheme presented in Algorithm 2. The generated code simply fires the enabled action whose guard is evaluated true at the time of scheduling. As discussed, *evaluatedGuard* provides the identifiers of actions whose guards are valid for particular iterations.

A particular action (i.e., either *act1* or *act2*) is executed in each iteration based on the activated guard. To this end, if the guard is true for *act1* (Line 20), a single token is consumed from *X*1 and a single token is consumed from *X*2 which is then written to the output *Y*1 (Lines 21–24). On the other hand, if the guard is true for *act2*, a token is consumed from *X*1, and two tokens are consumed from *X*2, where the first token of *X*2 is written to *Y*1, and the other to *Y*2 (Lines 27–32).

```
1  __kernel void split ( __global fifo_t* X1 ,
       __global fifo_t* X2, __global fifo_t* Y1 ,
        __global fifo_t* Y2, __global int*
       evaluatedGuard, int blockSize)
2  {
3    /* Declarations for All Inputs and Outputs */
4    __private int seq_X1[1];
5    __private int seq_X2[2];
6    __private int seq_Y1[1];
7    __private int seq_Y2[1];
8    int* x1_act1 = &seq_X1[0];
9    int* x1_act2 = &seq_X1[0];
10   int* x2_act1 = &seq_X2[0];
11   int* x2a_act2 = &seq_X2[0];
12   int* x2b_act2 = &seq_X2[1];
13   int* y1_act1 = &seq_Y1[0];
14   int* y1_act2 = &seq_Y1[0];
15   int* y2_act2 = &seq_Y2[0];
16   /* Generic Kernel Code */
17   int gid = get_global_id(0) * blockSize;
18   for (int x = 0; x < blockSize; x++) {
19     /* KPN Specific Kernel Code */
20     if(evaluatedGuard[x] == 0) {
21       fifoRead(X1, seq_X1, 1, gid);
22       fifoRead(X2, seq_X2, 1, gid);
23       *y1_act1 = *x2_act1;
24       fifoWrite(Y1, seq_Y1, 1, gid);
25     }
26     else if(evaluatedGuard[x] == 1) {
27       fifoRead(X1, seq_X1, 1, gid);
28       fifoRead(X2, seq_X2, 2, gid);
29       *y1_act2 = *x2a_act2;
30       *y2_act2 = *x2b_act2;
31       fifoWrite(Y1, seq_Y1, 1, gid);
32       fifoWrite(Y2, seq_Y2, 1, gid);
33     }
34   }
35 }
```

Listing 7 Generated kernel for *split* as illustrated in Fig. 10.

DDF Code Generator

In DDF, the decision on whether to consume tokens and to fire each action of a process is made dynamically at runtime when that particular process is triggered for execution. A process is simply scheduled by the host for execution if only one of its inputs has required tokens and only one of its outputs has required space. In contrast to SDF and KPN MoCs, the firing rules (including guards) of processes in DDF are completely evaluated within kernels at the device side. Each DDF kernel, therefore, must indicate to the host the number of tokens consumed/produced in each FIFO buffer for the dispatched execution instances. This fairly

complicates the code generation of kernels. In particular, the code generator incorporates a number of DDF MoC specific library functions designed to enable the dynamic evaluation of actions within kernels. The code generation based on the underlying semantics of the used DDF MoC is illustrated by the pseudo code given in Algorithm 3.

Algorithm 3: Pseudo code for DDF kernel code generation of a process p.

```
1  PeekTokens(inBuf(p)) ;
2  foreach action α in p do
3      if TokensAvailable(inGrd(α)) then
4          EvaluateGuard(γ_α);
5          if GuardValid(γ_α) &&
             TokensAvailable(inAct(α)\inGrd(α)) &&
             SpaceAvailable(outAct(α)) then
6              ConsumeTokens(inAct(α));
7              PerformComputations(α);
8              ProduceTokens(outAct(α));
9          end
10         else if GuardValid(γ_α) &&
             !(TokensAvailable(inAct(α)\inGrd(α)) &&
             SpaceAvailable(outAct(α))) then
11             WriteBuffersStatus(inAct(α))
12         end
13     end
14 end
```

First, the code is generated to peek tokens from all inputs for all actions of the process (Line 1). The algorithm then iterates through the modeled set of actions in the order of there definitions (Line 2). For each action, the algorithm proceeds as follows: First, the code is generated to check if enough tokens are available in the inputs used by the guard (Line 3). Second, code evaluating the guard is generated (Line 4). Next, the code is generated to fire an action (Lines 5–9). This involves code for checking if the guard is valid, the required number of tokens are available in all inputs and the required amount of space is available in all outputs (Line 5). The code is then generated for the enabled action (Lines 6–8) which involves code for consuming all inputs, performing modeled computations, and writing the computed results on the outputs. The code generator further generates code for the case if although the guard is true, however, either at least one of the inputs does not have enough tokens, or at least one of the outputs does not have enough space (Lines 10–12). In this case, code is generated to ascertain the status of tokens and space in each input and output FIFO buffer of the action, respectively. The status of each buffer in this respect is written and conveyed to the host using a data structure (Line 11). This case is typically used to indicate which input buffers lack the required number of tokens and/or which output buffers lack the required space. The generated kernel for a parallel process *POR* as shown in Fig. 11 is listed in Listing 8. For brevity, we illustrate here the generated code for the first two actions (*act*1 and *act*2).

OpenCL specific code segment. This code segment is exactly generated in the same way as generated in the case of the SDF MoC.

DDF MoC specific code segment. This code segment (Lines 14–47) is generated based on the scheme presented in Algorithm 3. First, the tokens are peeked from all inputs for all actions of the process (Lines 15–16). The generated code then simply evaluates each action for firing. If there are enough tokens available in the inputs used for guard (Lines 19 and 33), the guard is evaluated (Lines 20 and 34). If the guard is valid and the required number of tokens and the required amount of space is available in all input and output FIFO buffers, respectively, the action is fired (Lines 24–28 and 38–42). Depending on the availability of tokens/space and the enabled guard, either one or both of the actions (*act*1 and *act*2) can be executed in each iteration. In particular, if both actions are enabled, a token each is consumed from both inputs *X1* and *X2* (Lines 25 and 39) and a token each is produced to the output *Y* (Lines 26–27 and 40–41) by both actions. In case if the guard is enabled for one of the actions (say *act*1), however, there is no space available in the output *Y*, the status of tokens and space in *X1* and *Y*, respectively, are determined and written using the DDF MoC specific library function *writeStatus* (Lines 30–31).

Runtime System

The runtime system systematically employs OpenCL in the composition of the synthesis components to finally map and execute models based on different dataflow MoCs on COTS target hardware. It is typically designed in a centralized host and kernels architecture under the OpenCL abstraction as shown in Fig. 12. The host accommodates different essential components along with the *Runtime-Manager* that work together to implement low-level details such as: the schedulers, the communication mechanism, resource allocation, kernels mapping and handling etc.

Process and Device Queues

The *Process-Queue* is generated for the host at the back-end. Each element of this queue provides a special object of a process. Each object provides the specific attributes of the process to the host. This includes: the process name, the identified dataflow MoC, the associated FIFO buffers, the type of each buffer, and the process type. The queue, once generated, is maintained and updated by the host. In particular, the host assigns each object the process's status (idle, running or blocked) and the associated kernel.

Apart from the *Process-Queue* that is provided by the back-end, the host also generates a queue, namely the *Device-Queue*, using the OpenCL specification as depicted

in Fig. 12. The *Device-Queue* lists all the available devices of the target hardware. Each element of this queue provides a special object of a device. The device object provides an interface that is used by the host to access and to use the device for systematically executing kernels. In particular, each object features a command queue of a device, where the processes (kernels) can be mapped for execution. Each command queue can represent a complete device (e.g., a CPU) or even a compute unit of that device (e.g., a CPU-core).

```
__kernel void POR ( __global fifo_t* X1 ,
     __global fifo_t* X2, __global fifo_t* Y,
     int blockSize)
{
  /* Declarations for All Inputs and Outputs */
  __private bool seq_X1[1];
  __private bool seq_X2[1];
  __private bool seq_Y[1];
  bool* x1_act1 = &seq_X1[0];
  bool* x2_act2 = &seq_X2[0];
  bool* y_act1 = &seq_Y[0];
  bool* y_act2 = &seq_Y[0];
  /* Kernel Code */
  int gid = get_global_id(0) * blockSize;
  for (int x = 0; x < blockSize; x++) {
    /* Generate DDF Specific Kernel Code */
    bool ctrl_X1_act1 = fifoPeek(X1,seq_X1,1,
         gid);
    bool ctrl_X2_act2 = fifoPeek(X2,seq_X2,1,
         gid);
    bool ctrl_Y_act1 = fifoSpace(Y,0,Y->tail+
         gid);
    bool ctrl_Y_act2 = fifoSpace(Y,0,Y->tail+
         gid);
    if(ctrl_X1_act1){
      guard_act1 = (*x1_act1 == true );
      eval_impl_act1 = evalImplication(
           guard_act1, ctrl_X1_act1 &&
           ctrl_Y_act1);
    }
    else writeStatus(X1,1,X1->head+gid);
    if(eval_impl_act1 == '1'){
      fifoReadDDF(X1,4,X1->head+gid,1);
      *y_act1 = true;
      fifoWriteDDF(Y,seq_Y,1,gid);
    }
    else if(eval_impl_act1 == '0'){
      writeStatus(X1,1,X1->head+gid);
      writeStatus(Y,2,Y->tail+gid);
    }
    if(ctrl_X2_act2){
      guard_act2 = (*x2_act2 == true );
      eval_impl_act2 = evalImplication(
           guard_act2, ctrl_X2_act2 &&
           ctrl_Y_act2);
    }
    else writeStatus(X2,1,X2->head+gid);
    if(eval_impl_act2 == '1'){
       fifoReadDDF(X2,4,X2->head+gid,1);
       *y_act2 = true;
       fifoWriteDDF(Y,seq_Y,1, gid);
    }
    else if(eval_impl_act2 == '0'){
      writeStatus(X2,1,X2->head+gid);
      writeStatus(Y,2,Y->tail+gid);
    }
  }
}
```

Listing 8 Generated kernel for *POR* as illustrated in Fig. 11.

The Runtime-Manager

The core component of the runtime system is the *Runtime-Manager*, as shown in Fig. 12. It is a part of the host that uses the *Process-Queue* and the *Device-Queue*, and provides: the schedulers for scheduling processes based on the supported dataflow MoCs, the communication mechanism between the host and kernels, a dispatcher for mapping kernels to devices, and a handler mechanism for kernels using specialized callbacks.

Schedulers The schedulers are designed to schedule processes in the network based on the underlying dataflow MoC of each process. To support the execution of heterogeneous dataflow models characterized by different kinds of behaviors, the *Runtime-Manager* provides a two-level hierarchical scheduling scheme. At the first level, a baseline global scheduler is used that works in a dynamic round robin scheme. At the second level, specialized local schedulers are used, where each local scheduler is designed for a particular dataflow MoC. Altogether, the baseline global scheduler iterates through the *Process-Queue* in a round-robin fashion, and invokes the corresponding local scheduler for each process based on its underlying dataflow MoC. All the local schedulers are designed based on the dynamic data-driven scheduling schemes mainly because of the following reason: The target DPNs involve heterogeneous dataflow models consisting of static as well as dynamic behaviors. A common consistent dynamic environment is, therefore, needed to systematically schedule and execute heterogeneous models. Nevertheless, for a fully SDF network only consisting of static processes, one can generate a static scheduler at compile time.

Baseline Global Scheduler: The *Runtime-Manager* employs a simple global scheduler typically designed to handle the invocation of the specialized local schedulers for scheduling heterogeneous dataflow models. The global scheduling scheme is depicted in Algorithm 4.

Algorithm 4: Pseudo code for the baseline global scheduler.

```
1  while true do
2      foreach process p in the network do
3          if ProcessStatusIdle(p) then
4              if MoC(p) == "SDF" then
5                  CallSDFScheduler(p);
6              end
7              else if MoC(p) == "KPN" then
8                  CallKPNScheduler(p);
9              end
10             else if MoC(p) == "DDF" then
11                 CallDDFScheduler(p);
12             end
13         end
14     end
15 end
```

As DPNs do not generally enforce any termination criteria, the global scheduler runs forever (Line 1). It works in a round-robin fashion and selects the next process in the *Process-Queue* that is not currently running, in particular, has no pending dispatched executions (Lines 2–3). In case if a process has a pending call i.e., all dispatched kernel instances are not completely executed, the next invocation of this process is delayed until the process is idle again. The scheduler simply invokes the local scheduler of the selected process according to the underlying dataflow MoC of each process.

In the following, we present the specialized local schedulers based on the supported dataflow MoCs.

SDF Scheduler: The scheduling scheme based on the SDF MoC is depicted in Algorithm 5. A process based on the SDF MoC always consumes and produces a fixed number of tokens in each execution. This greatly simplifies the scheduling, in particular, a process is simply scheduled for execution if there is enough data available in all input buffers (Line 2) and if there is enough space available in all output buffers (Line 3).

Algorithm 5: Pseudo code for the SDF scheduling scheme.

```
1 for process p with SDF MoC in the network do
2     if TokensAvailable(inBuf(p)) then
3         if SpaceAvailable(outBuf(p)) then
4             ScheduleForExecution(p)
5         end
6     end
7 end
```

KPN Scheduler: As discussed, the KPN MoC supports static as well as sequential behaviors. A process is only scheduled for execution if it is already known that the firing rules (including guard) of one of the actions are valid. The firing rules in a process are, therefore, evaluated at the time of scheduling. This relatively complicates the scheduling. The scheduling scheme based on the KPN MoC is illustrated by the pseudo code given in Algorithm 6.

Algorithm 6: Pseudo code for the KPN scheduling scheme.

```
1  for process p with KPN MoC in the network do
2      foreach action α in p do
3          if TokensAvailable(inGrd(α)) then
4              EvaluateGuard(γ_α);
5              if GuardValid(γ_α)) then
6                  if
                       TokensAvailable(inAct(α)\inGrd(α))
                   then
7                      if SpaceAvailable(outAct(α))
                       then
8                          ScheduleForExecution(p)
9                      end
10                     else
11                         BlockUntilSpaceAvailable(p)
                           break foreach loop
12                     end
13                 end
14                 else
15                     BlockUntilTokensAvailable(p);
16                     break foreach loop
17                 end
18             end
19         end
20     end
21 end
```

Since the KPN MoC supports processes having sequential behaviors, it schedules a process for execution by evaluating the firing rules (including guards) sequentially in a predefined order. The KPN scheduler iterates through the set of modeled actions in the order of their definitions where for each action, it works as follows: First, the scheduler checks if enough tokens are available in the input buffers used by the guard (Line 3). If enough tokens are available and if the guard is valid (Line 5), the remaining (non-guarded) input buffers of the action are checked for enough tokens (Line 6). Only if there are enough tokens available, the output buffers are checked for space (Line 7). If enough space is available, the action is finally scheduled for execution (Line 8). In case if one of the conditions does not meet, the process is blocked until that condition is fulfilled (Lines 11 and 15).

Since the host and generated kernels are independent components, the evaluation sequence or order needs to be extracted from modeled processes at compile time. The extracted sequence can be used by the KPN scheduler at runtime to schedule processes for execution. To this end, we propose a systematic way of extracting the evaluation sequence by introducing the *input–output tree wrapper* (IOT-wrapper). The IOT-wrapper wraps the exact information of inputs/outputs required to schedule a process in a standard tree structure, while taking into account the underlying semantics of the KPN MoC. For each process,

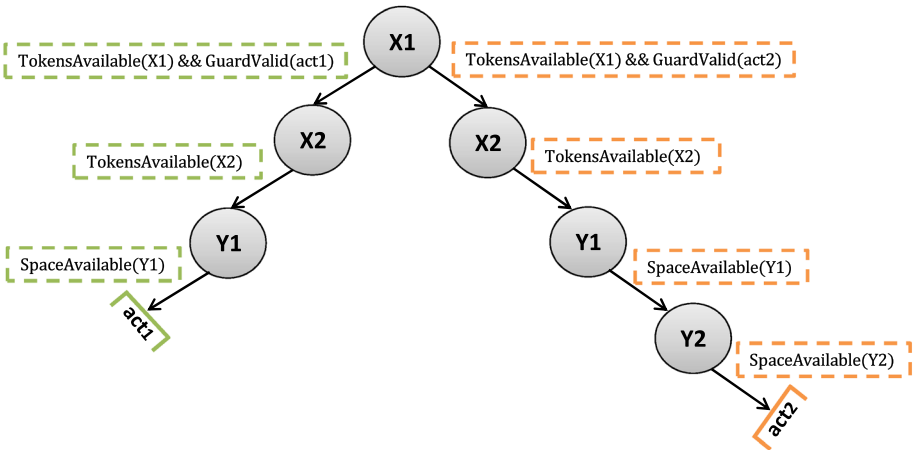

Fig. 13 Generated IOT-wrapper for *split* process as shown in Fig. 10: it consists of two branches where each branch corresponds to a particular action i.e., *act1* or *act2* of the process. The step functions of all nodes are illustrated in dashed boxes

a wrapper is generated at compile time from the modeled behavior. The IOT-wrapper generation based on the underlying KPN semantics is presented in [35].

The IOT-wrapper generated for a sequential process *split* as illustrated in Fig. 10, is shown in Fig. 13. The root node only involves the input *X1* as it is the only input used for guard by the process. The *StepFunction* generated and assigned to each node is shown in dashed boxes. The set of branches originating from the root node and extending up to the leaf node represents a particular action. For instance, *act2* is represented by branches originating from the root node ($X1$) and extending up to the leaf node ($Y2$).

KPN Scheduler based on IOT-wrapper: The KPN scheduler is provided with the generated IOT-wrappers of all processes in the used network. It uses a variant of the depth-first search (DFS) method [50] that starts at the root of the tree, selects a branch, and traverses through that branch as deep as possible until the leaf node is reached. In general, for each node, the scheduler calls the assigned *StepFunction*, and only moves to the next node if the function returns *true*. In particular, the *StepFunction* of the root node returns a number $num \in \mathbb{Z}$ mainly dependent on which guard is true. This number is used to select a specific branch originating from the root node that directs to a specific action whose guard is true. In case if the leaf node is reached and its *StepFunction* returns *true*, the scheduler triggers the process for execution. On the contrary, if the *StepFunction* of one of the nodes returns *false*, the process gets blocked until that node returns *true*.

Algorithm 7: Pseudo code for the DDF scheduling scheme.

```
1 for process p with DDF MoC in the network do
2     if TokensAvailableAnyInputFIFO(p) then
3         if SpaceAvailableAnyOutputFIFO(p) then
4             ScheduleForExecution(p);
5         end
6     end
7 end
```

DDF Scheduler: The DDF MoC evaluates the firing rules (including guard) of each action in a process as a part of the kernel at the device. This greatly simplifies the scheduling at the host and in fact the scheduler based on the DDF MoC is the simplest of all supported dataflow MoCs. The scheduling scheme based on the DDF MoC is illustrated by the pseudo code given in Algorithm 7. It follows an optimistic scheduling strategy that expects the firing of actions even if there is data available in only one input buffer and if there is space available in only one output buffer of the process. A process is therefore simply scheduled by the host for execution if there is enough data available in at least one of the input buffers (Line 2) and if there is enough space available in at least one of the output buffers (Line 3).

With that, we illustrated all the specialized schedulers based on the supported dataflow MoCs. In the following, we present two of the main components of the *RuntimeManager*, namely the *dispatcher* and the *handler*.

Dispatcher In general, a dispatcher is a special program which comes into play after the scheduler. When the scheduler completes its job of selecting a process, it is the dispatcher that gives a process control over the target device. The runtime system of the proposed framework also provides a special dispatcher built under the OpenCL abstraction. The global scheduler evokes one of the local schedulers based on the dataflow MoC of a process. The evoked scheduler fetches a ready process from the *Process-Queue* and provides it to the dispatcher as depicted in Fig. 14. The dispatcher acquires the device object from the *Device-Queue* and finally maps the fetched process on the command queue of the target device. The generated kernel of the dispatched process is then executed based on the used dataflow MoC.

Communication Mechanism and Handlers The proposed framework implements the FIFO buffers as bounded circular ring buffers. In general, this design enables the buffers to work as if the memory is contiguous and circular in nature. The communication between the host and kernels is realized using OpenCL memory objects (buffers). For each bounded FIFO buffer, an OpenCL buffer is created with the same design and size of the FIFO buffer. Each process object provided by the *Process-Queue* stores and links the address of each FIFO buffer with the associated OpenCL buffer. During the execution of a process, i.e., when the kernel of a process is being executed, data is read/written from/to the associated OpenCL buffers. When all the instances of the kernel are executed, i.e., the dispatched process executions are completed, the *Runtime-Manager* is then automatically notified to update the components. For that purpose, a handler mechanism is developed using callbacks as shown on the left part of Fig. 14.

The *Runtime-Manager* generates a callback interface each, for every existing device in the *Device-Queue* during the initialization of the queue. Based on the used dataflow MoC, the *Runtime-Manager* provides the MoC specific implementations for each generated callback interface. As a result, the dataflow MoC specific handler mechanism is invoked. The dispatcher sets up a callback event for each fetched process and links it with the callback handler of the device where it is dispatched. Hence, the completion of the kernel of the dispatched process automatically invokes the callback handler of the used device. The callback handler performs a set of general tasks including: retrieving data from the kernel (OpenCL buffers), updating all the FIFO buffers of the process, updating the status of the process, updating the *Process-Queue*, updating the device's load, and finally updating the OpenCL buffers. However, updating the FIFO buffers is a dataflow MoC specific task, and is therefore managed differently for the supported dataflow MoCs. Based on the SDF MoC, the data rate of a process remains fixed in each execution, and therefore each FIFO buffer is simply updated based on the specified static data rate. Based on the KPN MoC, since the processes are only scheduled if there exists one enabled action, each FIFO buffer is simply updated based on the enabled actions of dispatched instances. On the contrary, based on the DPN MoC, the processes are evaluated for their firing rules within kernels. Therefore, the amount of data consumed and produced is first measured at the host (handler) using the status of buffers, and finally each FIFO buffer of the process is updated accordingly.

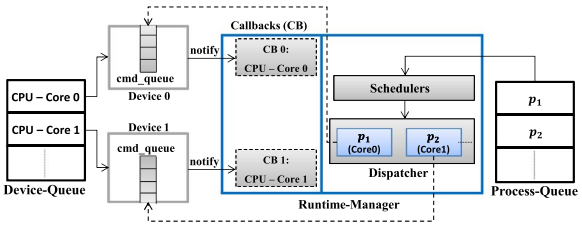

Fig. 14 The dispatching and handling mechanism of the *Runtime-Manager*. The dispatcher provided with the scheduled process maps the associated kernel on the command queue of the target device. The handler is mainly responsible for updating the runtime components at the host after the dispatched executions are performed

Experimental Evaluation

We organized our experimental evaluations into two parts: The first part features simple standalone benchmarks that perform simple operations and are especially designed to evaluate and compare the homogeneous versions generated by all individual supported dataflow MoCs for each benchmark (when possible). The second type presents a particular case study of the ConceptCar [37, 40] where different configurations of the ConceptCar's architecture are used to model and automatically generate implementations based on the individual dataflow MoCs as well as based on their heterogeneous combinations.

Experimental Setup

A variety of OpenCL supported devices have been employed for evaluation as listed in Fig. 15. The list involves five devices featuring three different device types from three different vendors. In particular, two different CPUs (**CPU1** and **CPU2**), one integrated GPU (**GPU1**) and two dedicated GPUs (**GPU2** and **GPU3**) featuring *Intel*, *AMD* and *NVIDIA* have been employed. The integrated GPU (**GPU1**) is built into the processor, and uses the system memory that is shared with the CPU (**CPU2**).

CPU1: Intel i5-8640U	
# physical cores	4
# logical cores	4
cache	6 MB Intel® Smart Cache
base frequency	3.20 GHz
RAM	8 GB
CPU2: Intel i7-8650U	
# physical cores	4
# logical cores	8
cache	8 MB Intel® Smart Cache
base frequency	1.90 GHz
RAM	8 GB
GPU1: Intel UHD Graphics 620	
# shaders/cuda cores	192
# compute units	24
cache	-
base clock	300 MHz
RAM	-
GPU2: AMD Radeon HD 5450 GPU	
# shaders/cuda cores	80
# compute units	2
cache	L1/L2: 8 KB/128 KB
base clock	650 MHz
RAM	512 MB GDDR2
GPU3: NVIDIA GeForce GTX 1050	
# shaders/cuda cores	640
# compute units	5
cache	L1/L2: 48 KB/1024 KB
base clock	1354 MHz
RAM	2 GB GDDR5

Fig. 15 The experimental setup: list of target devices employed to evaluate the proposed synthesis design flow. The specification of each target device is shown in the figure

In contrast, the dedicated GPUs (**GPU2** and **GPU3**) feature their own processors and their own source of memory.

For different target devices, different vendor-specific OpenCL implementations have been installed. This involves the software development kits (SDKs) from device vendors and the appropriate device drivers supporting OpenCL runtimes. The complete software environment used for executing generated versions on the target devices is summarized in Fig. 16.

Standalone Benchmarks

We designed a set of simple benchmarks consisting of processes having static, sequential or parallel functions. These benchmarks are therefore typically designed to offer a variety of processes having different kinds of behaviors that enable the evaluation and comparison of implementations based on all three different dataflow MoCs of the framework. Each benchmark only features processes based on a particular dataflow MoC. Each benchmark is organized in a network of three processes which are connected in a *producer-worker-consumer* setting. The *worker* process provides the main functionality of the benchmark and therefore performs the main operation. The designed standalone benchmarks along with their function types are listed in Fig. 17. A brief description of each benchmark is given as follows:

The sequential dynamic switch (*SeqDySwitch*) benchmark is designed to switch the only data input channel to any one of a number of individual output channels by the application of a control input. In contrast, the sequential dynamic worker (*SeqDyWorker*) benchmark performs the operation by taking one single input channel and copying its data to the only output channel based on the value of data of the only input channel. The sequential dynamic select (*SeqDySelect*) benchmark is a multiplexer that processes the information from multiple input channels into a single output channel by the application of a control input. It can simply be understood as a dynamic version of the if-then-else operation that sequentially consumes data from input channels based on the value of data on a control input. In contrast, the static if-then-else (*StITE*) benchmark is a static version of the

Device	Software Toolkit	Drivers
CPU1	Intel OpenCL SDK Version 6.3.0.1904	Intel processor driver version 10.0.1.9041.546
CPU2	Intel OpenCL SDK Version 6.3.0.1904	Intel processor driver version 10.0.1.9041.546
GPU1	Intel OpenCL SDK Version 6.3.0.1904	Intel graphics driver version 26.20.100.7639
GPU2	AMD OpenCL SDK version 3.0.130.135	AMD Radeon 5450 Driver Version 15.201.1151.1008
GPU3	CUDA Toolkit 10.1.243_426.00	NVIDIA Graphics Driver 451.67
Operating System		
Windows 10 Pro Version 1903 Build 18362.720		

Fig. 16 The software environment: list of software toolkits and drivers installed to enable the OpenCL implementations

		Generated Network		
Benchmark	Function Type	SDF	KPN	DDF
SeqDySwitch	sequential	×	✓	✓
SeqDyWorker	sequential	×	✓	✓
SeqDySelect	sequential	×	✓	✓
StITE	static	✓	✓	✓
StOR	static	✓	✓	✓
SeqDyMerge	sequential	×	✓	✓
SeqDySplit	sequential	×	✓	✓
ParDyOR	parallel	×	×	✓

Fig. 17 The designed standalone benchmarks. Each benchmark offers processes based on a particular kind of behavior as depicted by the function type. The right-hand side indicates the supported dataflow MoCs that were able to generate implementations for the particular benchmarks

if-then-else operation that always consumes data from all input channels in each execution. The sequential dynamic merge (*SeqDyMerge*) benchmark is designed to merge several input channels to a single common output channel by the application of a control input. In contrast, the sequential dynamic split (*SeqDySplit*) benchmark is designed to split a single input channel to a number of individual output channels by the application of a control input. The parallel dynamic OR (*ParDyOR*) benchmark performs the logical OR operation on two Boolean input channels and produces the result on the only output channel. It is a parallel version of the logical OR operation that can consume and produce tokens in parallel based on the availability of data on each input channel. In contrast, the static OR (*StOR*) benchmark is a synchronous version of the logical OR operation that always consumes tokens from both inputs in each execution. Apart from *ParDyOR* that incorporates the parallel function, all other benchmarks employ either static or sequential functions. In particular, apart from *StITE* and *StOR* which involve only static processes, all sequential benchmarks exhibit dynamic behaviors.

Each benchmark is modeled and automatically synthesized (when possible) based on all three supported dataflow MoCs of the framework. Thereby, three different implementations are automatically generated, namely the SDF MoC version, the KPN MoC version and the DDF MoC version. Since the SDF MoC only supports static behaviors, it could model and synthesize the *StITE* and *StOR* benchmarks. The KPN MoC supports both static and sequential functions and therefore able to model and synthesize all static and sequential benchmarks. However, it could not model and synthesize the only benchmark with the parallel function, namely the *ParDyOR* benchmark. The DDF MoC being the most generalized dataflow MoC of the lot supports static, sequential as well as parallel functions and therefore generated implementations for all designed benchmarks. The information regarding the implementations generated by the supported dataflow MoCs for the designed benchmarks is depicted in Fig. 17.

The generated versions for each benchmark based on different dataflow MoCs are evaluated based on their code size and the end-to-end performance. The code size for each generated version (implementation) of benchmark is measured as the sum of lines of code of all generated kernels for that version. The end-to-end performance, i.e., the total execution time of the network to process the complete input data set including initialization and termination of the program is considered as the comparison metric. The data set used has a maximum of ten thousand samples per input and the average of 50 repetitions is taken for each version.

Evaluation: Generated Code Size

The SDF MoC only supports static behaviors and therefore triggers a process when the data/space is available for all inputs/outputs. In each execution it consumes and produces statically determined fixed number of tokens from all inputs

and outputs, respectively. This simplifies the code generation and in particular generates very succinct kernel code for static processes. The KPN MoC supports static as well as sequential behaviors. Since processes in KPNs are sequential, their firing rules can be evaluated in a predefined order. It only triggers a process for execution when the exact information on inputs/outputs required to fire an action is available. This essentially simplifies the kernel code generation and therefore generates succinct kernel code for sequential processes. In contrast, the DDF MoC supports sequential as well parallel functions, and therefore dynamically evaluates actions including their inputs/outputs when the process is triggered for execution. Therefore, it accommodates additional code for enabling the dynamic evaluation of actions within kernels at runtime. The KPN MoC, therefore, generates more concise kernel code for sequential processes than the DDF MoC. The generated code size of each benchmark for the complete network based on all three dataflow MoCs is depicted in Fig. 18.

The additional kernel code overhead associated with the DDF MoC can therefore be observed from the number of lines of the generated code for each benchmark. For static benchmarks *StITE* and *StOR* that consist of only static processes, the SDF MoC generated the most succinct code of all generated versions. In particular, the generated code size based on the SDF MoC for *StITE* is about 83% and 3% lesser than the DDF and KPN versions, respectively. Similarly for *StOR*, the SDF version demonstrated a code reduction of about 90% and 3% in comparison to the DDF and KPN versions, respectively.

For all benchmarks consisting of sequential processes, the KPN MoC generated the most succinct code of all generated versions. For all sequential benchmarks, the KPN MoC generated at least 74% less lines of code than the DDF MoC. In particular, the biggest difference is recorded in *SeqDyWorker* where the generated code size based on the KPN MoC is 95% lesser than the DDF version. The *ParDyOR* benchmark features a parallel function and could only be modeled and synthesized based on the DDF MoC. The DDF MoC, therefore, offers a more generalized dataflow MoC that supports both sequential as well as parallel behaviors but at the cost of the additionally generated lines of kernel code.

	Lines of Generated Code		
Benchmark	SDF	KPN	DDF
SeqDySwitch	-	84	147
SeqDyWorker	-	63	123
SeqDySelect	-	79	140
StITE	81	83	148
StOR	70	72	133
SeqDyMerge	-	83	146
SeqDySplit	-	88	153
ParDyOR	-	-	159

Fig. 18 The generated code size for standalone benchmarks based on all three supported dataflow MoCs

Evaluation: The End-to-End Performance

Each generated version of a benchmark is either executed on **CPU1** (Intel) or **GPU2** (AMD) at a time to evaluate and compare the end-to-end performance of all used dataflow MoCs. On each target hardware, i.e., **CPU1** and **GPU2**, the average execution time of each generated version of a benchmark is measured against the number of data samples. The results are demonstrated for the maximum number of samples (i.e., ten thousand samples per input) where the biggest differences in execution times have been recorded as shown in Figs. 19 and 20.

Results: GPU2

The end-to-end performance of all generated versions of benchmarks on **GPU2** is shown in Fig. 19. As discussed for static benchmarks, the SDF MoC generated the most succinct kernel code. Considering the fact that the designed benchmarks only involve simple operations, the SDF MoC demonstrated the best end-to-end performance for *StITE* and *StOR*. In particular, the SDF version of *StITE* executed 1.74× faster than the DDF version and performed only slightly better than the KPN version. Similarly for *StOR*, the SDF version executed 1.59× faster than the DDF version and executed about 3% faster than the KPN version. Similarly, for all sequential benchmarks, the additional runtime overhead associated with the DDF MoC is propagated to the total execution time of the network resulting in elevated execution times. As a result, the KPN versions performed substantially better than the DDF versions. For all sequential benchmarks, the KPN versions executed at least 1.15× faster than the DDF versions. In particular, the biggest difference is observed in the case of *SeqDySelect* where the KPN version executed 2.87× faster than the DDF version.

Results: CPU1

In comparison to **GPU2**, the average execution time of each benchmark version is substantially reduced on **CPU1** as shown in Fig. 20. In contrast to OpenCL CPU where the host and the kernels reside on the same device, in the case of GPU, the data has to be transferred to the GPU and back to the main memory (host). This overhead therefore contributes

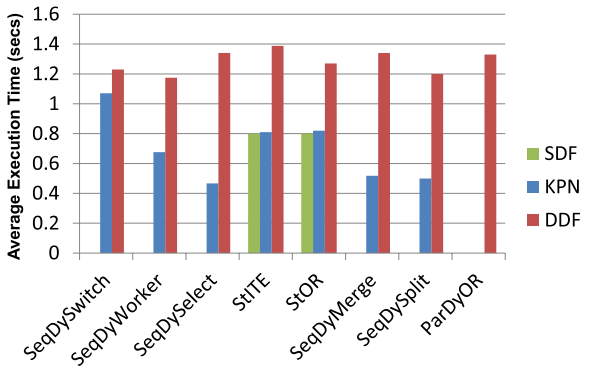

Fig. 19 End-to-end performance on **GPU2** for the generated versions of the standalone benchmarks. The results depicted in the figure are measured for 10K samples per input

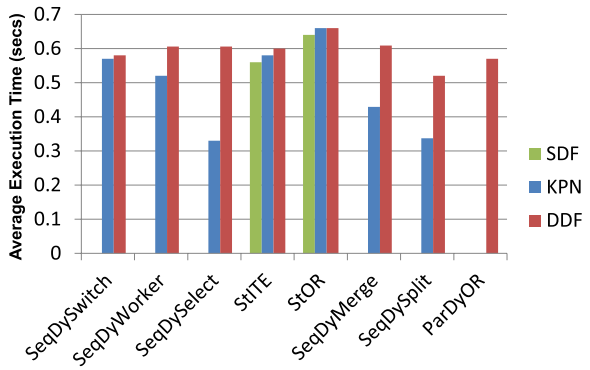

Fig. 20 End-to-end performance on **CPU1** for the generated versions of the standalone benchmarks. The results depicted in the figure are measured for 10K samples per input

in elevating the total execution time. On average, the generated versions on **CPU1** executed 1.75× faster than on **GPU1**. Similar to **GPU2**, the same trend of end-to-end performance has been observed on **CPU1**. The SDF MoC demonstrated the best end-to-end performance for static benchmarks *StITE* and *StOR*. The SDF version of *StITE* executed 7% and 3.5% faster than the DDF and KPN versions, respectively. Similarly for *StOR*, the SDF version performed slightly better than the DDF and KPN versions. For all sequential benchmarks, the KPN versions performed substantially better than the DDF versions. To this end, the KPN versions executed on average 1.4× faster than the DDF versions. In particular, the biggest difference is observed in the case of *SeqDySelect* where the KPN version executed 1.83× faster than the DDF version.

Results: Summary

The SDF MoC generated the most succinct kernel code and demonstrated the best end-to-end performance for simple static benchmarks. The KPN MoC performed significantly faster than the DDF MoC for all sequential benchmarks. The DDF MoC offers the most expressive semantics of all supported dataflow MoCs and therefore was able to generate implementations for all designed benchmarks. The DDF MoC enables one to model static, sequential as well as parallel behaviors but at the cost of additional runtime overhead. Thus, even for the simplest of the benchmarks, we observed that generating implementations based on the kind of behavior or the underlying dataflow MoC of each process results in substantially improved end-to-end performance. In other words, using a more generalized dataflow MoC for scheduling and executing rather restricted dataflow behaviors could result in inefficient system implementations.

Case Study: The ConceptCar's Dataflow Emulation

The *ConceptCar* [37, 40] (designed and developed by our group) is an experimental embedded system with the objective of testing and verifying car features by deploying different classes of applications. The ConceptCar, although not as big as a conventional car, has been built and engineered as close to a modern car as possible.

Hardware Design: The ConceptCar is a research platform remotely operated via a standard 2-channel (throttle and steering) 27 MHz radio transmitter system. It incorporates a set of sensors (wheel speed, gyro/accelerometer, distance etc.) for interacting with the environment and surroundings. It uses an air-cooled sensorless brushless electrical motor for driving, and a servo motor for steering. The power train of the ConceptCar features two independent power sources: one for the heavy load electric system (motors/actuators), and another one for powering up all the electronic control units (ECUs).

Computational Architecture: Although not incorporated with as many ECUs as a modern car can carry, the ConceptCar still features seven different ECUs, as shown in Fig. 21. These ECUs are organized in three processing units. The *SensorBoard* ECUs, as incorporated with different sensors, form the input processing unit which is responsible for interacting with the environment. The *multicore* ECU also known as *DataBoard* is used as a data processing unit and only comes into play when complex mathematical computations are required. The *ActorBoard* ECU forms the output processing unit and is responsible for creating the PWM signals to drive the actuators (dc motor and servo). The selector switch on *ActorBoard* chooses the source of data, either receiving processed data from *DataBoard* or normalized data from *SensorBoards*. Similar to a modern car, all

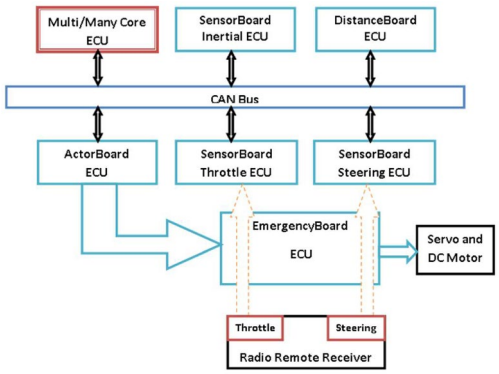

Fig. 21 Architecture of the ConceptCar

ECUs interact with each other via a centralized CAN bus architecture. Since the powertrain of the ConceptCar features two independent power sources, a special ECU called *EmergencyBoard* is integrated which separates the actuators from the other ECUs by galvanic isolation. *EmergencyBoard* therefore isolates functional sections of electrical systems to prevent current flow. This ECU only accepts the input from the radio receiver and *ActorBoard*, and bypasses it to *SensorBoards* and the actuators, respectively.

Dataflow Emulation: A dataflow emulation of the ConceptCar is devised where the operations of all the ECUs are emulated in a network of processes. The computations performed by each ECU are therefore modeled in a DPN process. This dataflow emulation allows us to produce two different test cases: The first test case as shown in Fig. 22 emulates the initial design of the ConceptCar without galvanic isolation i.e., the actuators are directly fed by *ActorBoard*. The second test case as shown in Fig. 27 considers the design with galvanic isolation provided by *EmergencyBoard*. For both test cases, the input data provided by the process *RadioRemoteReceiver* is collected from the centralized CAN bus and the results are validated against the logged outputs of the ConceptCar.

A number of implementations (versions) are automatically generated by the synthesis framework for both test cases where each test case is focused to evaluate particular aspects of synthesis. Each generated version is executed on three different devices, namely **CPU2** from Intel, **GPU1** from Intel and **GPU3** from NVIDIA as listed in Fig. 15. The target hardware for this case study, therefore, features three different types of devices involving a CPU, an integrated GPU and a dedicated GPU. Each generated version is evaluated for the resulting code size, the total network build time, and the end-to-end performance. The code size of each generated version is described as the sum of lines of code of all generated kernels. The network build time is defined as the total time taken by the OpenCL just-in-time (JIT) compiler to build all the kernels in the network and the MoC specific API functions of the used dataflow MoC(s). The build time is measured only by using **CPU2**. The end-to-end performance is defined as the total execution time of the network to process the complete input data set including initialization and termination of the program. The data set used has a maximum of five thousand samples per input and the average of 50 repetitions is taken for each version.

Test Case I: Open-Loop Configuration

The dataflow emulation of the design where *ActorBoard* directly feeds the actuators resulted in the open-loop configuration of the ConceptCar, as shown in Fig. 22. The original network features a heterogeneous DPN of different kinds of processes exhibiting static as well as sequential behaviors. The function or behavior type of each process is described at the bottom of each node as shown in Fig. 22. The main focus of this test case is to generate homogeneous implementations based on the individual dataflow MoCs of the framework and evaluate them for their resulting code size, the total network build time, and the end-to-end performance. The open-loop configuration is therefore modeled and automatically synthesized thrice, once for each individual dataflow MoC. Hence, three different versions are automatically generated by the synthesis framework, i.e., first based on the SDF MoC, second using the KPN MoC, and finally based on the DDF MoC. The original dataflow model of *ActorBoard* exhibits a sequential behavior as it utilizes data either from *DataBoard* or *SensorBoards* based on the input provided

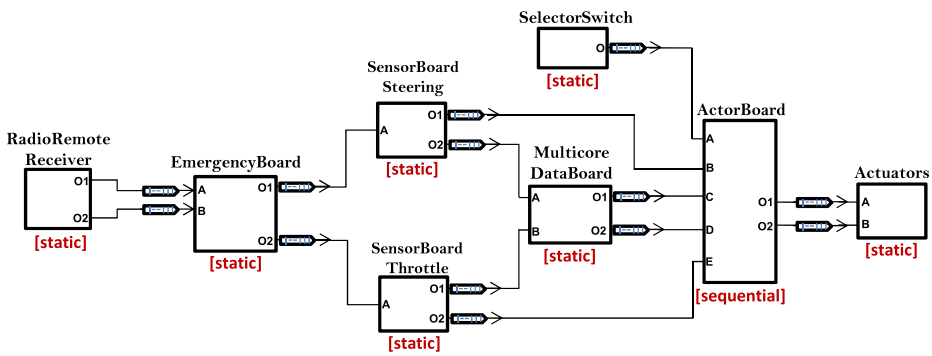

Fig. 22 The original dataflow network of the ConceptCar based on the open-loop configuration. As shown in the figure, the actuators are directly fed by *ActorBoard*. The behavior type of each process is described at the bottom of each node in square brackets

by *SelectorSwitch*. A static version of *ActorBoard* is also modeled that consumes data from all inputs in each execution. This allowed us to design a fully static network and to generate an implementation of the open-loop network based on the SDF MoC.

Generated Code Size and Network Build Time

The generated versions of the open-loop network based on all three dataflow MoCs are illustrated in Fig. 23. In particular, the generated kernel code size of each DPN process and the total build time for the complete network are given for each version.

The SDF MoC generated the most succinct kernel code for static processes. On the other hand, the KPN MoC generated the most concise code for sequential processes. In contrast to SDF and KPN MoCs, the DDF MoC offers a more flexible semantics where the decision on whether to consume/produce data in each execution is taken dynamically at runtime in the kernel code. Consequently, the generated DDF version accommodates additional kernel code for enabling the dynamic evaluation of actions at runtime when the process is triggered for execution. This overhead can therefore be observed from the number of lines of the generated code for each process and the total network build time. The generated code size of the DDF version for the complete network is 65% and 59% greater than the KPN and SDF versions, respectively. This results in an additional build time overhead of 391% and 384% in comparison to the build times of KPN and SDF versions, respectively. The overhead also reflects the additional time taken to build the DDF MoC specific API functions used for dynamic execution within kernels.

Finally, we also observed that the KPN version has slightly less code size than the SDF version for the complete open-loop network. This is mainly because the static version of *ActorBoard* consumes data from all inputs in each execution and therefore the corresponding generated kernel accommodated more lines of code. Precisely, the generated code for the static version of *ActorBoard* based on the SDF MoC is about 20% more than the dynamic version generated based on the KPN MoC.

End-to-End Performance

Each generated version of the open-loop network is executed on each target hardware at a time to evaluate and compare the end-to-end performance. On each target hardware. i.e., **CPU2**, **GPU1** and **GPU3**, the average execution time (in seconds) of each version is measured against the number of data samples as shown in Figs. 24, 25 and 26, respectively.

Regardless of which target hardware is used, the SDF version demonstrated the best end-to-end performance of

	Generated networks (open-loop)			
		SDF	DDF	KPN
Process size (lines of code)	Processes			
	RadioRemote	59	83	60
	SelectorSwitch	28	48	29
	EmergencyBoard	59	83	60
	SensorBoardSteering	102	171	105
	SensorBoardThrottle	102	171	105
	Databoard	62	87	64
	ActorBoard (static)	206	-	-
	ActorBoard (dynamic)	-	340	172
	Actuators	29	49	29
total		647	1032	624
Network build time (msecs.)		249	1205	245

Fig. 23 ConceptCar's open-loop setting: comparison of generated code size and network build time of all supported dataflow MoCs

all generated versions. Apart from *ActorBoard*, all processes in the original open-loop network are static. As already observed in the previous section, the SDF MoC generated the most succinct kernel code for static processes. Second, since the SDF MoC also simplifies the scheduling of processes, this further contributes to the improved end-to-end performance for the SDF version. The KPN version although offered a slightly less code size, however, induced a slight overhead in scheduling static processes in comparison to the SDF version. The SDF version, therefore, performed only slightly better than the KPN version, in particular, executed only 7% and 4.5% faster on **CPU2** and **GPU1**, respectively. On **GPU3** however the difference in performance is negligible.

In contrast to SDF and KPN MoCs, the additional runtime overhead associated with the DDF MoC is propagated to the end-to-end performance resulting in elevated execution times. Based on the results, as the number of samples increases, this effect induced by the overhead can be clearly observed. On **CPU2**, the DDF version took twice as much time as taken by the SDF version and took about 90% more time than the KPN version to execute the complete network for five thousand samples. On **GPU1**, the DDF version yielded about 145% and 135% more execution time than the SDF and KPN versions, respectively. Finally, on **GPU3**, the DDF version required 50% more time to process five thousand samples in comparison to the SDF and KPN versions. The DDF MoC although offers semantics to model sequential as well as parallel behaviors, but at the cost of the additional runtime overhead. Therefore, it exhibits a trade-off between expressiveness and overall performance.

All generated versions executed substantially faster on the CPU than on the used GPUs mainly because of the communication overhead associated with the OpenCL GPU. For

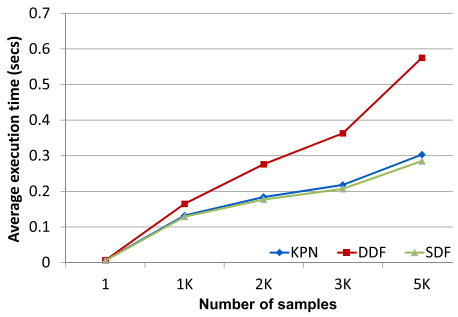

Fig. 24 Open-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **CPU2**

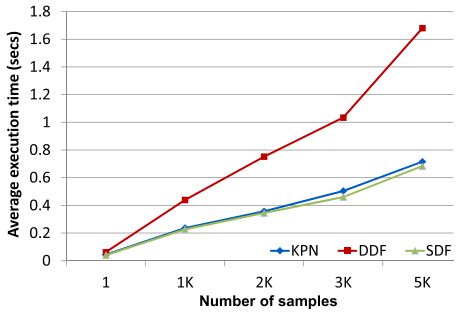

Fig. 25 Open-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU1**

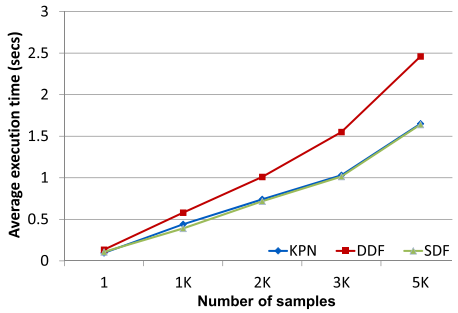

Fig. 26 Open-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU3**

instance, the SDF version on **CPU2** executed 4.75× and 1.4× faster than on **GPU3** and **GPU1**, respectively. Since, the integrated GPU (**GPU1**) share the same memory of the host, the versions executed substantially faster on **GPU1** than on the dedicated GPU (**GPU3**). For example, the SDF version on **GPU1** executed 1.39× faster than on **GPU3**.

Test Case II: Closed-Loop Configuration

The dataflow emulation of the design where the *EmergencyBoard* ECU separates the actuators from the rest of the ECUs by galvanic isolation resulted in the closed-loop configuration of the ConceptCar. The closed-loop setting therefore introduces a feedback loop in the network from *ActorBoard* into *EmergencyBoard* as shown in Fig. 27. The original network features a heterogeneous DPN of different kinds of processes exhibiting static, sequential and parallel behaviors. In particular, *EmergencyBoard* exhibits a parallel behavior as it features independent actions operating on the independent sets of inputs from *RadioRemoteReceiver* and *ActorBoard* and producing data to the independent sets of outputs. This test case focuses on observing how the feedback loop in the network affects the performances of the individual homogeneous implementations of all supported dataflow MoCs. Second, and most importantly, it also demonstrates how the proposed synthesis method effectively exploits the heterogeneity by generating implementations based on the underlying dataflow MoC of each process to further improve the end-to-end performance.

The closed-loop configuration is modeled and automatically synthesized four times, once for each individual dataflow MoC and once based on the heterogeneous combination of all dataflow MoCs. Hence, four different versions are automatically generated by the synthesis framework, i.e., first based on the SDF MoC, second using the KPN MoC, third based on the DDF MoC, and finally the heterogeneous version based on the combination of all used dataflow MoCs. Since, the original dataflow model of *EmergencyBoard* exhibits a parallel behavior, a static version is also designed that consumes data from all inputs in each execution. This allowed us to generate implementations of the closed-loop network based on the SDF and KPN MoCs.

Generated Code Size and Network Build Time

The generated versions of the closed-loop network are illustrated in Fig. 28. The generated homogeneous versions based on the individual dataflow MoCs demonstrated the same pattern in code size as observed in the case of the open-loop network. The generated code size of the DDF version for the complete network is 67% and 61% greater than the KPN and SDF versions, respectively. This resulted in an additional build time overhead of 385% and 379% in comparison to the build times of KPN and SDF versions, respectively.

The heterogeneous version is automatically generated based on the kind of behavior or the underlying dataflow MoC of each process in the network. The generated code size of the heterogeneous version is only slightly greater than the KPN and SDF versions. Since the heterogeneous

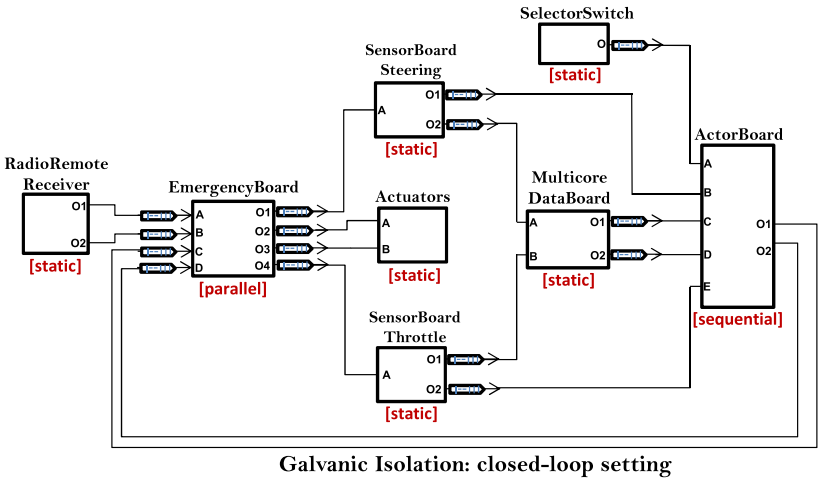

Fig. 27 The original dataflow network of the ConceptCar based on the closed-loop configuration. As shown in the figure, a feedback loop is introduced into the network from *ActorBoard* into *EmergencyBoard*. The behavior type of each process is described at the bottom of each node in square brackets

	Generated networks (closed-loop)				
		SDF	DDF	KPN	Hetero
Process size (lines of code)	Processes				
	RadioRemote	59	83	60	59
	SelectorSwitch	28	48	29	28
	EmergencyBoard (static)	86	-	87	-
	EmergencyBoard (parallel)	-	139	-	139
	SensorBoardSteering	102	171	105	102
	SensorBoardThrottle	102	171	105	102
	Databoard	62	87	64	62
	ActorBoard (static)	206	-	-	-
	ActorBoard (dynamic)	-	340	172	172
	Actuators	29	46	29	29
total		674	1085	651	693
Network build time (msecs.)		252	1208	249	348

Fig. 28 Closed-loop setting: comparison of generated code size and network build time of all supported dataflow MoCs including their heterogeneous combination

version employed the MoC specific API functions of all the used dataflow MoCs, this resulted in the build time overhead of about 40% and 38% in comparison to the build times of KPN and SDF versions, respectively. However, the code size of the DDF version is 56% greater than the heterogeneous version and therefore required 247% more time to build the complete network.

End-to-End Performance

Each generated version of the closed-loop network is executed on each target hardware at a time to evaluate and compare the end-to-end performance. On each target hardware. i.e., **CPU2**, **GPU1** and **GPU3**, the average execution time (in seconds) of each version is measured against the number of data samples as shown in Figs. 29, 30 and 31, respectively.

Regardless of which target hardware is used, it can be observed that the introduction of the feedback loop in the network elevates the execution times of the SDF and KPN versions to an unacceptable level. As discussed, the original dataflow model of *EmergencyBoard* exhibits a parallel behavior. Since, the SDF MoC only supports static behaviors, a static version of *EmergencyBoard* is designed to generate the SDF and KPN versions. The static version of *EmergencyBoard* therefore requires data in all inputs before it can be scheduled for execution. With a feedback loop introduced into *EmergencyBoard*, the SDF and KPN versions only schedule and communicate a single execution at a time for all processes (except *RadioRemoteReceiver*) at the device. Consequently, this induces a lot of scheduling and communication overhead between the host and device, and, therefore, resulted in excessively elevated execution times. The DDF version on the other hand employs a parallel version of *EmergencyBoard* and therefore attempts to schedule and communicate as many executions at a time as possible based on the availability of data on independent sets of inputs. Thus, even with the associated runtime overhead, the DDF version outperformed the SDF and KPN versions.

On **CPU2**, the DDF version executed 31× and 41× faster than the SDF and KPN versions, respectively, in executing the complete network for five thousand samples. Similarly, the end-to-end performance of SDF and KPN versions on both the used GPUs reached to an unacceptable level. On **GPU1**, the SDF executed 66× and the KPN version executed 149× slower in comparison to the DDF version. On **GPU3**, the SDF and KPN versions are 51× and 45×, respectively, slower than the DDF version. The difference in execution times is bigger on the GPUs than on the CPU mainly because of the communication overhead associated with OpenCL GPUs. The SDF version performed substantially better than the KPN version on the CPU and the integrated GPU, in particular, executed about 1.33× and 2.2× faster, respectively. This is mainly because the KPN version

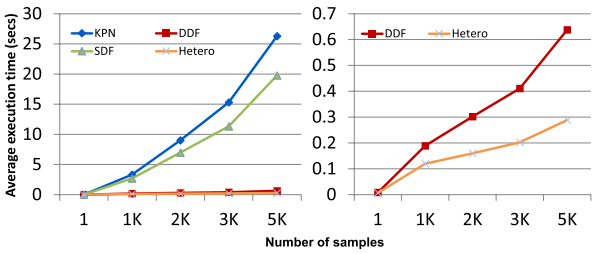

Fig. 29 Closed-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **CPU2**. The right-hand side graph particularly shows the performance comparison between the DDF version and the heterogeneous version

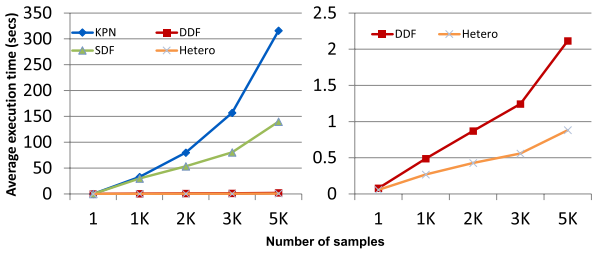

Fig. 30 Closed-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU1**. The right-hand side graph particularly shows the performance comparison between the DDF version and the heterogeneous version

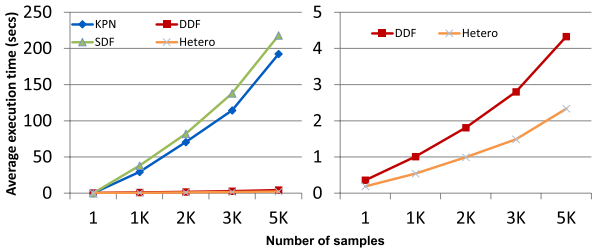

Fig. 31 Closed-loop setting: comparison of end-to-end performance of all supported dataflow MoCs on **GPU3**. The right hand side graph particularly shows the performance comparison between the DDF version and the heterogeneous version

induced an overhead in scheduling static processes for a large number of single executions. Interestingly, the KPN version executed about 1.13× faster than the SDF version on the dedicated GPU. The KPN version only dispatches a process for execution on the device if there exists one of the actions whose firing rules are satisfied. It therefore evaluates the firing rules at the scheduling time on the host side. This relaxes the computation on the device side. Since the dedicated GPU uses its own CPU and memory, the execution of simpler operations on less powerful processing cores contributed in the improved end-to-end performance for the KPN version.

The heterogeneous version: The main highlight of this test case is to evaluate the ability of the proposed synthesis method to efficiently exploit the heterogeneity of different kinds of behaviors of processes in the network. The heterogeneous version is therefore automatically generated based on the underlying dataflow MoC of each process in the network. Regardless of which device is used for execution, the heterogeneous version demonstrated a substantial improvement in performance in comparison to the most efficient homogeneous version of the closed-loop network. The performance comparison between the heterogeneous version and the DDF version is especially shown on the right-hand side graphs of Figs. 29, 30 and 31. In particular, the heterogeneous version demonstrated a speedup of 2.2×, 2.4× and 1.86× in comparison to the DDF version on **CPU2**, **GPU1** and **GPU3**, respectively. This speedup achieved by the heterogeneous version on all target devices is illustrated in Fig. 32. The speedup is calculated with reference to the computation time (in seconds) of the generated homogeneous DDF version. The heterogeneous version therefore significantly improved the performance by eliminating the overhead induced by the feedback loop in the SDF and KPN versions, and by avoiding the additional runtime overhead of the DDF version.

Hence, it can be concluded from this particular test case that the ability to exploit the heterogeneity of different kinds of behaviors of processes in DPNs contributes in efficient system implementations with substantially improved end-to-end performance.

Summary

The first test case featured an open-loop network which is mainly designed to generate and evaluate homogeneous implementations based on all individual supported dataflow MoCs. Considering the fact that most of the processes in the network exhibit static behaviors, the SDF version demonstrated the best end-to-end performance of all generation versions. The DDF version induced the runtime overhead of evaluating actions within kernels and performed the slowest of all generated versions. In particular, the DDF version took about 145% and 135% more execution time than the SDF and KPN versions, respectively.

The second test case introduced a feedback loop from *ActorBoard* to *EmergencyBoard* that resulted in a closed-loop network. We observed that the introduction of the feedback loop in the network greatly degraded the end-to-end performance of SDF and KPN versions. In particular, the DDF version even with the associated runtime overhead outperformed the SDF and KPN versions. However, the heterogeneous version that exploited the heterogeneity of different kinds of

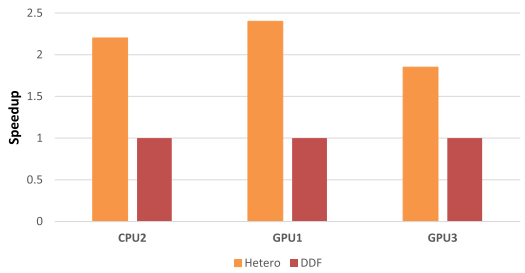

Fig. 32 Closed-loop setting: speedup gained by the heterogeneous version on all target devices. The speedup is calculated with reference to the computation time (in seconds) of the generated homogeneous DDF version

processes significantly improved the performance, in particular, executed up to 2.4× faster than the fastest homogeneous DDF version. Hence, the ability of the proposed synthesis method to exploit heterogeneity in a DPN effectively contributed in achieving the best end-to-end performance.

Conclusions

This paper presented the automatic software synthesis of systems based on three different well-defined dataflow MoCs including their heterogeneous combinations. We proposed a synthesis design flow that offers a comprehensive tool chain including specialized code generators and the runtime system for the supported dataflow MoCs. First, this allowed us to meet the objective of validating, evaluating and comparing the artifacts exhibited by different dataflow MoCs at the implementation level under the shed of a common design tool. Second, an efficient and smarter synthesis method is presented that targets and exploits heterogeneity in dataflow networks by generating implementations based on the kinds of behaviors of the processes. Finally, this work also tackled the challenge of systematically handling the portability of systems on COTS heterogeneous platforms.

Based on our evaluations, we observed that even for the simplest of the benchmarks, the generated versions based on the kinds of behaviors of the processes demonstrated the best end-to-end performance. In particular, using a more generalized dataflow MoC for scheduling and executing rather restricted dataflow behaviors resulted in inefficient system implementations. Based on our case study, we observed that the heterogeneous versions generated by the proposed synthesis method demonstrated a substantial improvement in performance. In particular, the heterogeneous versions demonstrated up to 2.4× speedup than the most efficient generated homogeneous version. Based on the evaluations, it can be concluded that the ability to exploit the heterogeneity of different kinds of behaviors of processes in DPNs contributes in efficient system implementations with substantially improved end-to-end performance.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Bai Y. Model-based design of embedded systems by desynchronization. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, 2016. PhD.
2. Bai Y, Rafique O and Schneider K. A model-based design flow for asynchronous implementations from synchronous specifications. In: Design, automation and test in Europe (DATE). Grenoble, France, 2021. IEEE Computer Society, pp. 862–7.
3. Bai Y, Schneider K, Bhardwaj N, Katti B and Shazadi T. From clock-driven to data-driven models. In: Formal methods and models for codesign. Lausanne, Switzerland, 2014. IEEE Computer Society, pp. 32–41.
4. Benveniste A, Caillaud B, Le Guernic P. From synchrony to asynchrony. In: Baeten JCM, Mauw S, editors. Concurrency theory (CONCUR), vol. 1664. LNCS. Eindhoven: Springer; 1999. p. 162–77.
5. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R. The synchronous languages twelve years later. Proc IEEE. 2003;91(1):64–83.
6. Berry G, Gonthier G. The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program. 1992;19(2):87–152.
7. Bhattacharyya SS, Eker J, Janneck JW, Lucarz C, Mattavelli M, Raulet M. Overview of the MPEG reconfigurable video coding framework. J Signal Process Syst. 2011;63(2):251–63.
8. Boutellier J, Wu J, Huttunen H, Bhattacharyya SS. PRUNE: dynamic and decidable dataflow for signal processing on heterogeneous platforms. IEEE Trans Signal Process. 2018;66(3):654–65.
9. Brooks CX, Lee EA, Tripakis S. Exploring models of computation with Ptolemy II. In: Givargis T, Donlin A, editors. International conference on hardware/software codesign and system synthesis (CODES+ISSS). Scottsdale: ACM; 2010. p. 331–2.

10. Buck JT. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, University of California, Berkeley, California, USA, 1993. PhD.
11. Dennis JB. First version of a data-flow procedure language. In: Robinet B, editor. Programming symposium, vol. 19. LNCS. Paris: Springer; 1974. p. 362–76.
12. Didier K, Potop-Butucaru D, Iooss G, Cohen A, Souyris J, Baufreton P, Graillat A. Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware. ACM Trans Archit Code Optim. 2019;16(3):24:1-24:27.
13. Eker J and Janneck JW. CAL language report. ERL Technical Memo UCB/ERL M03/48, EECS Department, University of California at Berkeley, Berkeley, California, USA, 2003.
14. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y. Taming heterogeneity—the Ptolemy approach. Proc IEEE. 2003;91(1):127–44.
15. Engels M, Bilsen G, Lauwereins R and Peperstraete J. Cyclo-static dataflow. In: International conference on acoustics, speech and signal processing (ICASSP). Detroit, Michigan, USA, 1995. IEEE Computer Society, pp. 3255–8.
16. Falk J, Haubelt C, Teich J, Zebelein C. SysteMoC: a data-flow programming language for codesign. Netherlands: Springer; 2017. p. 59–97.
17. Faustini AA. An operational semantics for pure data flow. In: Nielsen M, Schmidt EM, editors. International colloquium on automata. Languages and programming (ICALP), volume 140 of LNCS. Århus: Springer; 1982. p. 212–24.
18. Geilen M, Basten T. Requirements on the execution of Kahn process networks. In: Degano P, editor. European symposium on programming (ESOP), vol. 2618. LNCS. Warsaw: Springer; 2003. p. 319–34.
19. Girault A. A survey of automatic distribution method for synchronous programs. In: Synchronous languages, applications, and programming (SLAP). Edinburgh, Scotland, UK, 2005, pp. 1–20. (**unpublished workshop proceedings**)
20. Haubelt C, Falk J, Keinert J, Schlichter T, Streubühr M, Deyhle A, Hadert A, Teich J. A SystemC-based design methodology for digital signal processing systems. EURASIP J Embed Syst. 2007;2007(1):15–15.
21. Kahn G and MacQueen DB. Coroutines and networks of parallel processes. In: Gilchrist B, editor, Information processing. North-Holland, 1977; pp. 993–8.
22. Kahn G. The semantics of a simple language for parallel programming. In: Rosenfeld JL, editor. Information processing. Stockholm: North-Holland; 1974. p. 471–5.
23. Karp RM, Miller RE. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J Appl Math (SIAP). 1966;14(6):1390–411.
24. Kuhn T, Forster T, Braun T and Gotzhein R. FERAL—framework for simulator coupling on requirements and architecture level. In: Formal methods and models for codesign. Portland, OR, USA, 2013; pp. 11–22. IEEE Computer Society.
25. Lee JH, Nigania N, Kim H, Patel K and Kim H. OpenCL performance evaluation on modern multicore CPUs. Sci Programm. 2015; pp. 859491:1–859491:20
26. Lee EA, Messerschmitt DG. Synchronous data flow. Proc IEEE. 1987;75(9):1235–45.
27. Lee EA, Parks T. Dataflow process networks. Proc IEEE. 1995;83(5):773–801.
28. Lund W, Kanur S, Ersfolk J, Tsiopoulos L, Lilius J, Haldin J and Falk U. Execution of dataflow process networks on OpenCL platforms. In: Euromicro international conference on parallel, distributed, and network-based processing. Turku, Finland, 2015; pp. 618–25. IEEE Computer Society.
29. Parks TM. Bounded scheduling of process networks. PhD thesis, Princeton University, 1995. PhD.
30. Parks TM, Pino JL and Lee EA. A comparison of synchronous and cyclo-static dataflow. In: Asilomar conference on signals, systems and computers. Washington, District of Columbia, USA, 1995; pp. 204–210. IEEE Computer Society.
31. Potop-Butucaru D, Caillaud B, Benveniste A. Concurrency in synchronous systems. Formal Methods Syst Design (FMSD). 2006;28(2):111–30.
32. Rafique O and Schneider K. A model-based synthesis framework for the execution of dynamic dataflow actors. In: International conference on internet of things embedded systems and communications. Hammamet, Tunisia, 2018. IEEE Computer Society.
33. Rafique O and Schneider K. Employing OpenCL as a standard hardware abstraction in a distributed embedded system: a case study. In: Conference on cyber-physical systems and internet-of-things. Budva, Montenegro, 2020; pp. 1–7. IEEE Computer Society.
34. Rafique O and Schneider K. Evaluating OpenCL as a standard hardware abstraction for a model-based synthesis framework: a case study. In: International conference on model driven engineering and software development. Prague, Czech Republic, 2019.
35. Rafique O and Schneider K. Integrating Kahn process networks as a model of computation in an extendable model-based design framework. In: International conference on model driven engineering and software development. SCITEPRESS, 2021.
36. Rafique O and Schneider K. SHeD: a framework for automatic software synthesis of heterogeneous dataflow process networks. In: Euromicro conference on digital system design (DSD), Portorož, Slovenia, 2020. IEEE Computer Society.
37. Rafique O, Gesell M, Schneider K. Generating hardware specific code at different abstraction levels using Averest. In: Corporaal H, Stuijk S, editors. International workshop on software and compilers for embedded systems (SCOPES). Sankt Goar: ACM; 2013. p. 90–2.
38. Rafique O, Bai Y, Schneider K and Yan G. Efficient implementation of heterogeneous dataflow models using synchronous IO patterns. In: Euromicro conference on digital system design (DSD). Palermo, Sicily, Italy, 2021; pp. 82–9. IEEE Computer Society.
39. Rafique O, Bai Y, Schneider K and Yan G. Synthesis of heterogeneous dataflow models from synchronous specifications. In: Computers, software, and applications conference (COMPSAC). Virtual Conference, 2021. IEEE Computer Society.
40. Rafique O, Gesell M and Schneider K. Learning various aspects of a distributed real-time automotive embedded system. In: Marwedel P, Jackson J and Ricks K (eds) Workshop on embedded and cyber-physical systems education (WESE), Montreal, Canada, 2013. ACM.
41. Rafique O, Schneider K. Automatic software synthesis of static and dynamic dataflow process networks. In: International workshop on interplay of model-driven and component-based software engineering. Munich, Germany; 2019.
42. Sander I, Jantsch A, Attarzadeh-Niaki S-H. ForSyDe: system design using a functional language and models of computation. In: Ha S, Teich J, editors. Handbook of hardware/software codesign, chapter 4. Berlin: Springer; 2017.
43. Schneider K. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
44. Schor L, Bacivarov I, Rai D, Yang H, Kang S-H, Thiele L. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Jerraya A, Carloni L, Mooney V, Rabbah R, editors. Compilers, architecture, and synthesis for embedded systems (CASES). Tampere: ACM; 2012. p. 71–80.
45. Schor L, Tretter A, Scherer T and Thiele L. Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL. In: IEEE symposium on embedded systems for real-time multimedia. IEEE Computer Society, 2013; pp. 41–50.

46. Shen J, Fang J, Sips H, Varbanescu AL. An application-centric evaluation of OpenCL on multi-core CPUs. Parallel Comput. 2013;39(12):834–50.
47. Stone JE, Gohara D, Shi G. OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng. 2010;12(3):66–73.
48. Stuijk S, Geilen M, Theelen BD, Basten T. Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications. In: Carro L, Pimentel AD, editors. International conference on embedded computer systems: architectures, modeling, and simulation (SAMOS). Samos: IEEE Computer Society; 2011. p. 404–11.
49. Stuijk S, Geilen M and Basten T. SDF3: SDF for free. In: Application of concurrency to system design (ACSD). Turku, Finland, 2006; pp. 276–8. IEEE Computer Society.
50. Tarjan R. Depth first search and linear graph algorithms. SIAM J Comput (SICOMP). 1972;1(2):146–60.