Automatic Software Synthesis of Static and Dynamic Dataflow Process Networks

Omair Rafique and Klaus Schneider
Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany
{rafique,schneider}@cs.uni-kl.de

***Abstract*—A general dataflow process network (DPN) consists of a network of actors that communicate with each other via statically determined point-to-point buffers. While the general model of computation (MoC) does not impose further restrictions, different classes of DPNs with precise MoCs have been introduced over time. These classes differ in the kinds of behaviors of the actors which affects on the one hand the expressiveness of the DPN class as well as the methods for their analysis and synthesis. A combination of particular classes of DPNs can be effectively used to model and synthesize heterogeneous parallel systems. There are design tools for synthesis that can be conveniently used for implementing individual classes of DPNs, however, they do not support the use of the combination of these DPNs. We therefore envisage a common model-based design flow that allows the modeling of a system based on a combination of particular classes of DPNs, and automatically synthesizes them for heterogeneous architectures. In this paper, we mainly present in detail how the proposed design flow facilitates the automatic synthesis of two individual classes of DPNs by using different code generators for the particular kinds of actors. We therefore first describe the proposed classes of DPNs and then present a common automatic synthesis that details the code generators and the runtime system. Finally, we demonstrate the use of our framework with a simple test case that has been modeled and synthesized individually for each employed class of DPN.**

***Index Terms*—model-based synthesis, models of computation, dataflow programming paradigm**

I. Introduction

A model of computation (MoC) precisely determines *why, when and which atomic action of a system is executed.* Dataflow process networks (DPN) [1], [2] can be used to define such MoCs. In general, a DPN consists of a network of autonomous process nodes (actors) that communicate with each other via unidirectional First-In-First-Out (FIFO) buffers. While the general model of computation does not impose further restrictions, many different classes of DPNs [3]–[5] have been introduced. Each class defines a specific MoC by specifying a particular execution and communication semantics. Based on that, these classes differ in the kinds of behaviors of the actors which determines the expressiveness of the DPN class as well as the methods for their analysis (predictability) and synthesis (efficiency).

A heterogeneous computing system can accompany different devices including single-core and multi-core processors with application-specific hardware. At the level of its software architecture, it may consist of many components concurrently running on these devices that interact with each other via particular MoCs. To develop such systems using DPNs, a heterogeneous combination of particular classes of DPNs can be effectively used. There exist design tools for modeling like Ptolemy [6] and FERAL [7] that support the use of a combination of particular MoCs for the modeling and simulation of heterogeneous architectures. However, the existing design tools for synthesis generally incorporate a particular MoC, usually restricted to a specific class of DPN. Each framework is therefore dedicated to demonstrate the artifacts exhibited by a specific MoC. For instance, a framework based on the synchronous dataflow (SDF) can be conveniently used for modeling and implementing synchronous behaviors. Similarly, the others with a more generalized DPN MoC allow one to capture more flexible behaviors. In order to model and implement heterogeneous behaviors based on the combination of these DPNs, a common synthesis design flow is needed.

We therefore envision a common model-based design flow that supports the modeling of a system based on a heterogeneous combination of particular classes of DPNs, and automatically synthesizes them to implementations for cross-vendor heterogeneous architectures. In a nutshell, we envision a model-based automatic synthesis that implements models using a combination of particular MoCs on heterogeneous hardware. Based on this overall vision, in this paper, we present in detail how the proposed synthesis design flow integrates different individual classes of DPNs in a common framework. The method mainly focuses on the software synthesis of two different classes of DPNs that uses different code generators for the particular kinds of actors. In general, efficiently synthesizing DPNs for different architectures is a challenging task as the architecture specification and the runtime for each target hardware has to be taken into account. To this end, the common abstraction provided by the open computing language (OpenCL) [8] for cross-vendor heterogeneous architectures can be effectively used for implementing different classes of DPNs. The proposed framework employs a subset of the CAL actor language (CAL) [9] to model behaviors, and logically uses OpenCL as an operating system (OS) to implement modeled behaviors on any OpenCL abstracted target hardware.

In summary, we make the following contributions in this paper:

- We present a model-based design flow that allows us to model and synthesize two different classes of DPNs,

namely the static (synchronous) dataflow (SDF) and the dynamic dataflow (DDF).
- Using a formal description of a general dataflow network, we elaborate the specific interpretations of the proposed MoCs.
- We present the individual code generators of the proposed SDF and the DDF actors.
- We demonstrate the applicability of our framework with a simple test case, modeled and synthesized twice, once based on the SDF and second using the DDF.
- We present experimental results to analyze and compare the code size, the total network build time, and the total execution time of synthesized implementations.

Based on the stated contributions, this paper mainly emphasizes on the modeling and software synthesis of behaviors based on individual classes of DPNs. Nevertheless, it also provides the basis for the future work to support behaviors based on heterogeneous combination of these DPNs.

II. Background

This section first discusses in general the dataflow process networks (DPNs) and their synthesis, and then presents the related synthesis frameworks.

A. Dataflow Process Networks

A dataflow process network (DPN) [1], [2] models a system as a directed graph that consists of nodes (actors) and edges (FIFO buffers). Actors can be viewed as concurrent processes that perform computations and exchange data only through the unidirectional FIFO buffers. Each actor performs a computation by firing, where it consumes data tokens from its input buffers and produces data tokens for its output buffers. The firing of an actor is generally triggered by the availability of data.

Although the general model of computation (MoC) does not impose any further restrictions, many different classes of DPNs have been introduced over time [3]–[5], [10]. Each class specifies a particular execution and communication semantics that governs the firing of actors and the communication over FIFO buffers. To this end, these classes mainly differ based on how each actor triggers an execution, and based on how each actor execution consumes/produces data, i.e., either fixed or dynamic. Based on these factors, these classes can be categorized into static and dynamic ones.

Static DPNs are generally characterized as the ones where the number of tokens produced/consumed by each actor on each execution is specified statically at compile time, and hence actors can be scheduled statically at compile time. On the one hand, these characteristics allow powerful design-time analysis techniques (e.g. for predictability and decidability), but on the other hand they limit the expressiveness by excluding dynamic behaviors (like select and switch nodes). Examples of static DPNs are: synchronous dataflow (SDF) [4], homogeneous synchronous dataflow (HSDF) [4] and cyclo-static dataflow (CSDF) [3].

Dynamic DPNs are more general in that the order in which actors are triggered for execution, as well the number of tokens produced/consumed by each actor on each execution is decided dynamically at runtime. This allows conditional or data dependent executions of actors, in particular, each actor can produce and consume different numbers of tokens in every firing. This generalization results in higher expressiveness and flexibility, but makes the analysis more difficult. Examples of dynamic DPNs are: Kahn process networks (KPN) [11], Boolean dataflow (BDF) [5], and dynamic dataflow (DDF) [10].

B. Synthesis of Different Classes of DPNs

As discussed, the existing classes of DPNs differ in the kinds of behaviors of the actors which affects on the one hand the expressiveness of the DPN class as well as the methods for their analysis and synthesis. The design tools for modeling like Ptolemy [6] and FERAL [7] support the modeling and simulation of behaviors based on various MoCs, including different classes of DPNs. These frameworks provide a common platform to compose different models under the supervision of software components called *directors*. On the contrary, the design tools for synthesis are so far limited to specific classes of DPNs i.e., each tool is dedicated to a particular class of DPN. Each framework therefore allows one to model and synthesize behaviors based on a specific MoC, i.e., the underlying class of DPN. To this end, a design tool that only supports a static class of DPN, can be conveniently used for the modeling and synthesis of static (synchronous) behaviors. Similarly, a design tool based on a dynamic class of DPN, can be conveniently employed for dynamic and asynchronous behaviors. However, in order to model and synthesize heterogeneous behaviors based on the combination of particular classes of DPNs, a common synthesis design flow is still needed that can support these DPNs.

We therefore propose the idea of a common automatic synthesis design flow that allows us to model behaviors based on particular classes of DPNs, and automatically synthesizes them for heterogeneous architectures, mainly by using a combination of different code generators. To this end, this work presents in detail how the proposed approach supports the synthesis of two different classes of DPNs, namely the SDF from static DPNs and the DDF from dynamic DPNs. This paper mainly focuses on the individual code generators, however, also forms the basis for the future work to use the combination of these generators to support heterogeneous DPNs.

C. Related Frameworks

In this section, we present a few examples of dataflow oriented synthesis frameworks:

In [12], the HW/SW co-design methodology based on CAL is built as an Eclipse plug-in on top of ORCC [13]. The final implementations provided by this framework are based on a dynamic DPN.

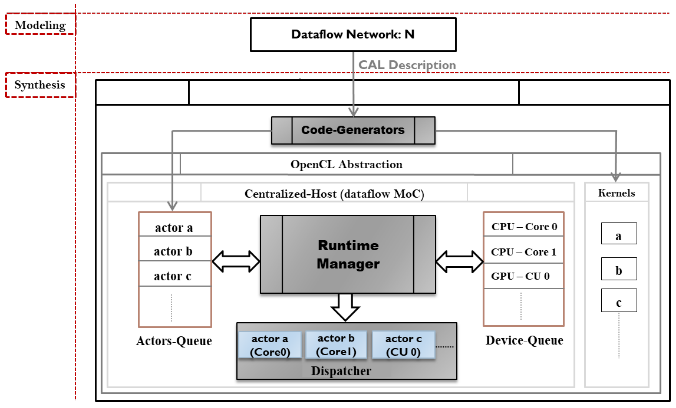


Fig. 1: The basic building block diagram of the framework.

The DAL framework [14] presents a scenario-based design flow for mapping streaming applications onto heterogeneous systems. Behaviors are modeled based on Kahn process networks (KPNs) [11] and a finite state machine.

The framework presented in [15] introduces a design flow for executing applications specified as synchronous dataflow (SDF) graphs on heterogeneous systems using OpenCL. Similarly, the work presented in [16] provides an approach to translate behaviors modeled with CAL into programs running some of the computations on OpenCL. The methodology incorporates static analysis and confined to the synthesis of behaviors modeled with the SDF MoC.

Another dataflow oriented framework [17], [18] proposes a MoC as a symmetric-rate dataflow, a restriced form of SDF, where the token production rate and the token consumption rate per FIFO channel is symmetric.

Therefore, the existing frameworks based on DPNs generally provide a single dedicated code generator for supporting synthesis of behaviors based on a particular MoC.

III. The Proposed Design Flow

A. Overview

The proposed design flow is systematically organized in two stages of modeling and synthesis, as shown in Fig. 1. At first, the modeling stage employs a subset of the CAL actor language (CAL) [9] to model behaviors. CAL is a dataflow language that allows one to model behaviors as DPNs. The main reason for choosing CAL is that it does not refer to any particular class of DPN, and instead offers an abstract notion of an actor to model a behavior based on various classes of DPNs [19]. Therefore, it is conveniently employed by the framework to model behaviors based on the proposed classes of DPNs (detailed in Section III-B). Since a general discussion of CAL is out of scope for this paper, we refer to [9], [19] for further details.

For the synthesis stage, the framework logically incorporates the OpenCL specification as an operating system (OS) mainly because of two reasons: First, it provides an abstraction for heterogeneous hardware, and second, the framework uses this abstraction in the composition of the synthesis where different components implement different low-level details. To this end, OpenCL distinguishes between a host and kernels where the host is a centralized entity that is connected to one or more compute devices (CPU, GPU etc.) and is responsible for the execution of kernels. Kernels are C-like functions that actually implement the abstract behavior of the system or part of the system. Therefore, the framework adopts this idea of hosts and kernels for the synthesis as shown in Fig 1. Since a general discussion of OpenCL is out of scope for this paper, we refer to [8] for further details.

Overall, the modeled behavior based on any proposed class of DPN is provided to the synthesis stage. The synthesis stage incorporates an individual code generator for each DPN class (detailed in Section III-C1), and uses a centralized host (detailed in Section III-C2) that provides different components (including the runtime system) to finally implement the modeled behavior based on the underlying DPN class (MoC) on the targeted OpenCL-abstracted hardware.

B. The Proposed Models of Computation

In this section, we first formally describe a general dataflow network based on the used subset of CAL, which is then used to specify the execution semantics of the proposed classes of DPNs, namely the SDF MoC and the DDF MoC.

1) Formal Description of the Proposed Model: A dataflow network is an an ordered pair $\aleph = (F, A)$, consisting of FIFO buffers F and actors A. $F \subseteq (\{i, o\} \times \mathbb{N})$ are the available input and output FIFO buffers. Every DPN has a finite number of FIFO buffers F. Input FIFO buffers are defined by $F_{input} = F \cap (\{i\} \times \mathbb{N})$, and output FIFO buffers are defined by $F_{output} = F \cap (\{o\} \times \mathbb{N})$. A is the finite set of actors. Each actor $(F_{in}, F_{out}, G_A) \in A$ also has a subset of input FIFO buffers $F_{in} \subseteq F_{input}$, a subset of output FIFO buffers $F_{out} \subseteq F_{output}$, and an associated set of guarded-actions G_A. The input and output FIFO buffers of an actor are always mutually exclusive, i.e, $F_{in} \cap F_{out} = \phi$. In general, each DPN actor has firing rules that determine when enough tokens are available to enable that actor. Also, a computation an actor can perform is determined by a function that consumes a finite number of input tokens and produces a finite number of output tokens. To support the modeling of different kinds of behaviors (e.g., synchronous and asynchronous), we further decompose and organize these firing rules and computations of actors in a set of what we call guarded-actions G_A. Each guarded-action g_a $(F_i, F_o, \vec{\mathcal{V}}_i, \vec{\mathcal{V}}_o, \gamma, \tau) \in G_A$ has a subset of input FIFO buffers $F_i \subseteq F_{in}$, a subset of output FIFO buffers $F_o \subseteq F_{out}$, a set consisting of sequences of input token variables $\vec{\mathcal{V}}_i$ of F_i, a set consisting of sequences of output token variables $\vec{\mathcal{V}}_o$ of F_o, and two functions γ and τ.

Explanation. The code template for a generic actor 'a' is listed in Listing 1: An actor 'a' with *'m'* inputs and *'n'* outputs is declared with a finite set of input FIFO buffers (F_{in_1} to F_{in_m}) and a finite set of output FIFO buffers (F_{out_1} to F_{out_n}) with a data type α (Line 1). The behavior of this generic actor is specified by a set of actions G_A, where each action g_a upon execution can consume input tokens, perform some computations on consumed tokens and produce output tokens. For brevity, we list a single example action designated by 'g_{a_1}' (Lines 3-12). The action is declared with a finite set of *'t'* input FIFO buffers $F_i = \{F_{i_1}, F_{i_2} ..., F_{i_t}\}$ (Line 3), and a finite set of *'l'* output FIFO buffers $F_o = \{F_{o_1}, F_{o_2}, ..., F_{o_l}\}$ (Line 4). Input FIFO buffers F_i are declared with their respective sequences of input token variables $\vec{\mathcal{V}_i} = \{\mathcal{V}_{i_1}, \mathcal{V}_{i_2}, ..., \mathcal{V}_{i_t}\}$. Each sequence $\mathcal{V}_{i_j} \in \vec{\mathcal{V}_i}$ then consists of a finite number of input token variables. For instance, if F_{i_j} has a sequence of *'p'* tokens per action execution, the corresponding set of local variables is defined by $\mathcal{V}_{i_j} = \{v_{i_{j\_1}}, v_{i_{j\_2}}, ..., v_{i_{j\_p}}\}$ (Line 3). The number of variables ('p' in this case) determines the token consumption rate of F_{i_j} per execution of an action. Similarly, output FIFO buffers F_o are declared with their respective sequences of output token variables $\vec{\mathcal{V}_o} = \{\mathcal{V}_{o_1}, \mathcal{V}_{o_2}, ..., \mathcal{V}_{o_l}\}$. Each sequence $\mathcal{V}_{o_j} \in \vec{\mathcal{V}_o}$ then consists of a finite number of output token variables. For instance, if F_{o_j} has a sequence of *'h'* tokens per action execution, the corresponding set of local variables is defined by $\mathcal{V}_{o_j} = \{v_{o_{j\_1}}, v_{o_{j\_2}}, ..., v_{o_{j\_h}}\}$ (Line 4). The number of variables 'h' denotes the token production rate of F_{o_j} per execution of this action.

Listing 1: Code template for a generic actor 'a'.

```
1  actor a() α F_in1, ... α F_inm ==> α F_out1, ... α F_outn :
2    //guarded−action example
3    g_a1: action F_i1 :[v_i1_1, v_i1_2 ... v_i1_p], ... F_it :[v_it_1, v_it_2 ... v_it_q]
4    ==> F_o1 :[v_o1_1, v_o1_2 ... v_o1_g], ... F_ol :[v_ol_1, v_ol_2 ... v_ol_h]
5    guard
6      v_i1_1 > 1 and v_it_q < 0 and v_it_1 ...
7    var
8      α v_o1_1, v_ol_h ...
9    do
10     v_o1_1 := v_i1_1*2 + v_i1_1*v_it_1 ...
11     ...
12   end
13   ...
14   g_ax:
15   ...
16 end
```

The function γ: $\mathcal{E}_\mathcal{B}(\mathcal{V}_\gamma) \rightarrow \mathcal{B}$ is a Boolean function of an action that evaluates a finite set of Boolean expressions $\mathcal{E}_\mathcal{B} = \{\mathcal{E}_{\mathcal{B}_1}, \mathcal{E}_{\mathcal{B}_2}, ..., \mathcal{E}_{\mathcal{B}_n}\}$ applied on the individual input token variables, to a Boolean ($\mathcal{B} = \{0, 1\}$), where $\mathcal{V}_\gamma \subseteq \bigcup \vec{\mathcal{V}_i}$. τ: $\vec{\mathcal{V}}_i^{|F_i|} \rightarrow \vec{\mathcal{V}}_o^{|F_o|}$ is a firing function of an action that upon firing consumes tokens $\vec{\mathcal{V}_i}$ from F_i and produces tokens $\vec{\mathcal{V}_o}$ to F_o.

Execution Conditions. There exist two **minimal conditions** (m_c) for any action g_a to fire. **(1)** There should be sufficient input tokens $\vec{\mathcal{V}_i}$ available to bind F_i to appropriate values. **(2)** There should be sufficient room for the output tokens $\vec{\mathcal{V}_o}$ in their respective output FIFO buffers F_o. For an action g_a that requires input tokens to have particular values, an additional condition can be specified using a **guard** (Line 5). A guard consists of a set of Boolean expressions ($\mathcal{E}_\mathcal{B}$) that are applied on a set of individual input token variables $\mathcal{V}_\gamma$ (Line 6). The combined evaluation of $\mathcal{E}_\mathcal{B}$ is represented by the function γ. The computations an action can perform in the form of the firing function τ are defined within **do/end** blocks (Lines 9-12).

Restrictions. Based on the used subset of CAL, we also consider following restrictions: **(1)** The actions (G_A) are always evaluated for execution sequentially in the same order of their definitions. **(2)** Generally, F_i and F_o can be overlapping across different actions, however, guard-conditions are always exclusive. This ensures that for each execution of an actor, the actions will never compete for an execution for any set of data values (tokens).

TABLE I: Action implication $(\gamma, m_c) \Rightarrow V$

γ	m_c	$(\gamma, m_c) \Rightarrow V$
1	1	'1'
1	0	'0'
0	1	'x'
0	0	'x'

2) The SDF Execution Semantics: The SDF MoC [4] allows one to model static (synchronous) behaviors. It is a restricted class of DPN in the sense that each actor has a constant consumption/production token rate per execution (data rate) as well as the topology of data flow across actors (data path) is fixed. For modeling static behaviors using the proposed model, an additional restriction is considered in that F_i and F_o across actions are always unique.

Each actor ($\in A$) in $\aleph$ is triggered for an execution if and only if sufficient input data is available for all its guarded-actions and sufficient space is available for the outputs of those actions. The data rate and the data path of an actor in each execution remain the same. Upon an execution, for each guarded-action ($\in G_A$) of an actor, $\vec{\mathcal{V}_i}$ is consumed, and guard is evaluated using the function γ. In case if guard holds, the corresponding action is executed which performs a computation, and finally produces an output $\vec{\mathcal{V}_o}$ of that action based on the firing function τ. Overall, each actor execution consumes a fixed number of tokens from all input FIFO buffers (F_{in}), and produces a fixed number of tokens for all output FIFO buffers (F_{out}). Hence, the proposed SDF MoC is intrinsically deterministic.

3) The DDF Execution Semantics: The DDF MoC [10] allows one to model dynamic and asynchronous actors. To this end, each actor ($\in A$) in $\aleph$ is triggered for an execution if there is input data available for any input FIFO buffer and if space is available for any output FIFO buffer of that actor. The data rate and the data path of an actor can change per execution depending on which action is executed. At first, for all actions G_A of an actor, $\vec{\mathcal{V}_i}$ is peeked from F_i, and F_o is checked for space based on $\vec{\mathcal{V}_o}$. Next, for each $g_a \in G_A$ of an actor, guard is evaluated using the function γ, and an additional implication $(\gamma, m_c) \Rightarrow V$ is evaluated, where γ represents a combined evaluation of all Boolean expressions, and m_c represents the evaluation of minimal conditions to fire an action as discussed in Section III-B1. Based on that, this implication that forms the basis of making decisions dynamically at runtime about whether to execute an action and consume/produce data tokens, is evaluated based on the conditions as shown in Table I. In case if the implication is evaluated to one, the corresponding action is executed which consumes the peeked tokens from F_i, performs a computation, and finally produces output tokens in F_o of that action. On

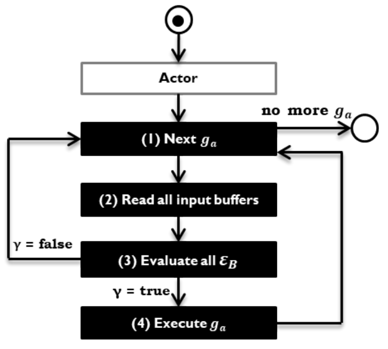


Fig. 2: The algorithm for the SDF MoC based code generation.

the contrary, if the implication is evaluated to zero i.e., if at least one of the minimal conditions is false, the peeked tokens from F_i are not consumed and the action is not executed. Finally, if γ is evaluated as false for an action, the implication is evaluated to a don't-care state 'x'.

As each actor upon execution is permitted to check F_{in} by peeking before it can be finally consumed, thus allows a non-blocking read. This makes the proposed MoC more flexible, however, may lead to non-deterministic behaviors e.g., a non-determinate merge.

Listing 2: SDF based generated code of an action from the actor 'a' as illustrated in Listing 1.

```
1   /*g_a-by-g_a execution:*/
2   /* g_{a_1} /*
3   /*step1: read all inputs for an action*/
4   fifoRead(F_{i_1}, buf_F_{i_1}, p, gid, &cnt_F_{i_1});
5   v_{i_{1_1}} = buf_F_{i_1}[gid*p + cnt_F_{i_1}.current_count];
6   ...
7   fifoRead(F_{i_t}, buf_F_{i_t}, q, gid, &cnt_F_{i_t});
8   v_{i_{t_q}} = buf_F_{i_t}[gid*q + cnt_F_{i_t}.current_count];
9   /*step 2: evaluate guard expressions E_B*/
10  guard_v_{i_{1_1}} = (expression) ? true: false;
11  ...
12  guard_v_{i_{t_q}} = (expression) ? true: false;
13  /*step 3: execute guarded-action*/
14  if(guard_v_{i_{1_1}} && guard_v_{i_{t_q}} && ...) {
15      /*do end*/
16      v_{o_{1_1}} = v_{i_{1_1}}*2 + v_{i_{1_1}}*v_{i_{t_1}} ... ;
17      bytes = fifoWrite(F_{o_1}, V_{o_1}, ..., gid, &cnt_F_{o_1});
18      ...
19  }
```

C. Synthesis

As discussed, the synthesis stage uses different components, as shown in Fig 1, and finally implements the modeled behavior based on the underlying DPN class (MoC) on the targeted OpenCL-abstracted hardware. This section explains in detail different components of the synthesis stage.

1) Code Generators: Each code generator is a core component of the framework that generates code strictly based on the semantics of the underlying DPN class and the OpenCL specification. Based on the OpenCL paradigm, the code generator supplied with a CAL description generates an OpenCL kernel for each actor of the network, as shown in Fig 1. This section explains the code generation of the kernels based on the proposed classes of DPNs, namely the SDF and the DDF, as were introduced in Section III-B.

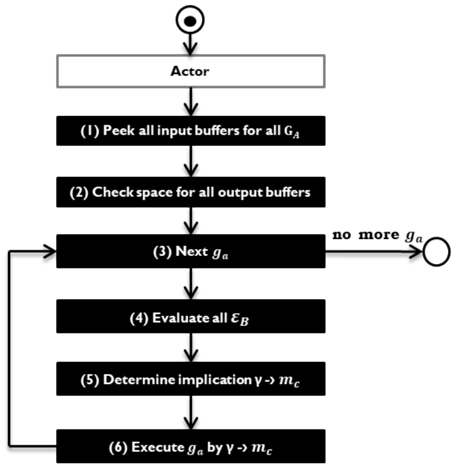

Actor
(1) Peek all input buffers for all G_A
(2) Check space for all output buffers
no more g_a
(3) Next g_a
(4) Evaluate all $\mathcal{E}_B$
(5) Determine implication γ -> m_c
(6) Execute g_a by γ -> m_c

Fig. 3: The algorithm for the DDF MoC based code generation.

a) Code Generation SDF: The algorithm for the code generation based on the proposed SDF MoC is shown in Fig. 2. Assuming that the generic actor 'a' as illustrated in Listing 1, is a synchronous (static) one, the generated code based on this algorithm is depicted in Listing 2. For brevity, we only show the generated code for the example action 'g_{a_1}' from Listing 1.

(1) The code generator sequentially iterates through available actions (G_A) of an actor, where for each g_a it proceeds as follows: **(2)** It generates code that read (consume) all the input tokens $\vec{\mathcal{V}}_i$ from F_i. For this purpose, it inserts a **fifoRead** for each input FIFO buffer for reading tokens, and generates code that assign each token to its corresponding variable (Lines 3-8). **(3)** Next, the code generator generates code for evaluating all the Boolean expressions $\mathcal{E}_{\mathcal{B}}$ of a guard (Lines 9-12). **(4)** Finally, it generates code that evaluate the combined Boolean result (Line 14), and if true, execute the corresponding action where all the defined computations are performed including writing output tokens to the associated output FIFO buffers (Lines 15-18). For each output FIFO buffer, a **fifoWrite** is inserted for writing tokens based on the specified output token variables $\mathcal{V}_{o_j}$. Overall, the code generator follows a g_a-by-g_a execution scheme until there is no more g_a left.

b) Code Generation DDF: The algorithm for the DDF MoC based code generation is shown in Fig. 3. Using this algorithm, the generated code for the generic actor 'a' as illustrated in Listing 1, is listed in Listing 3. For brevity, we only show the generated code related to the example action 'g_{a_1}' from Listing 1.

(1) The code generator first generates code that peek all the input tokens $\vec{\mathcal{V}}_i$ for all actions from all the input FIFO buffers F_{in}. To this end, for each input FIFO buffer, it inserts a **fifoPeek** (Line 2), and generates code that assign each token to its corresponding token variable (Line 3). **(2)** It then generates code that check if there is sufficient space available in each output FIFO buffer based on the number of output token variables. For this purpose, a **CTRLBITDynamic** with a parameter "get" is used for each output FIFO buffer (Lines 5-7).

(3) The code generator then proceeds in a g_a-by-g_a execution scheme until there is no more g_a left. For each g_a in an actor, the code generator works as follows: **(4)** It generates code that evaluate all the Boolean expressions $\mathcal{E}_{\mathcal{B}}$ of a guard (Lines 10-12). **(5)** Next, it generates code that determine the implication $(\gamma, m_c) \Rightarrow V$. For this purpose, the **evalImplication**(γ, m_c) is inserted that finally evaluates the implication (Lines 13-14). **(6)** Next, the code generator generates code that execute an action based on the evaluated implication from **(5)**. To this end, the generated code uses the results of the implication to decide whether to execute an action or not (Lines 15-27). If the final result evaluates to '0', it implies that although γ holds true, at least one of the input/output FIFO buffers of that action does not have sufficient data/space available. For this purpose, a **CTRLBITDynamic** with a parameter "auto" is used (Line 17), that determines whether each input/output FIFO buffer has sufficient data/space or not, and updates the buffer accordingly. On the contrary, if the final result evaluates to '1' (Line 20), it implies that an action is ready to execute. To this end, the corresponding action is executed where all the defined computations are performed including writing output tokens to the associated output FIFO buffers (Lines 21-26). For each output FIFO buffer, a **fifoWrite** is used for writing tokens based on the specified output token variables $\mathcal{V}_{o_j}$ (Line 23). Similarly for each input FIFO buffer, a **CTRLBITDynamic** with a parameter "clear" is used to update the buffer for consumed tokens (Line 25).

Listing 3: DDF based generated code of an action from the actor 'a' as illustrated in Listing 1.

```
1   /*peek all input ports for all actions. Returns ctrl bits automatically*/
2   ctrl_$\mathcal{V}_{i_1}$ = fifoPeek($F_{i_1}$, buf_$F_{i_1}$, p, gid, &cnt_$F_{i_1}$);
3   $v_{i_{1\_1}}$ = buf_$F_{i_1}$[gid*p + cnt_$F_{i_1}$.current_count];
4   ...
5   /*all output ctrl bits for space check*/
6   ctrl_$\mathcal{V}_{o_1}$ = CTRLBITDynamic($F_{o_1}$, "get", $F_{o_1}$->tail + gid, &cnt_$F_{o_1}$, g);
7   ...
8   /*$g_a$-by-$g_a$ execution:*/
9   /* $g_{a_1}$/*
10  /*step 1: evaluate guard expressions $\mathcal{E}_{\mathcal{B}}$ and determine action implication*/
11  guard_$v_{i_{1\_1}}$ = (expression) ? true: false;
12  ...
13  ev_impl_g_a_1 = evalImplication(guard_$v_{i_{1\_1}}$ && guard_$v_{i_{t\_q}}$ ...,
14  ctrl_$\mathcal{V}_{i_1}$ && ctrl_$\mathcal{V}_{i_t}$ && ctrl_$\mathcal{V}_{o_1}$ && ctrl_$\mathcal{V}_{o_l}$);
15  /*step 2: execute $g_a$ by $(\gamma, m_c) \Rightarrow V$ (in the order: '0', '1')*/
16  if(ev_impl_g_a_1 == '0'){
17      CTRLBITDynamic($F_{i_1}$, "auto", $F_{i_1}$->head + gid, &cnt_$F_{i_1}$, p);
18      ...
19  }
20  if(ev_impl_g_a_1 == '1'){
21      $v_{o_{1\_1}}$ = $v_{i_{1\_1}}$*2 + $v_{i_{1\_1}}$*$v_{i_{t\_1}}$ ... ;
22      ...
23      bytes = fifoWrite($F_{o_1}$, $\mathcal{V}_{o_1}$, ..., gid, &cnt_$F_{o_1}$);
24      ...
25      CTRLBITDynamic($F_{i_1}$, "clear", $F_{i_1}$->head + gid, &cnt_$F_{i_1}$, p);
26      ...
27  }
```

2) Centralized Host: As shown in Fig. 1, the centralized host of the framework is further composed of essential components that work together for implementing low-level details such as the scheduling policy, the communication mechanism, resource allocation, etc. One of such components is a queue of actor objects denoted as *Actors-Queue*, generated by the code generator for the host. The *Actors-Queue* contains a special

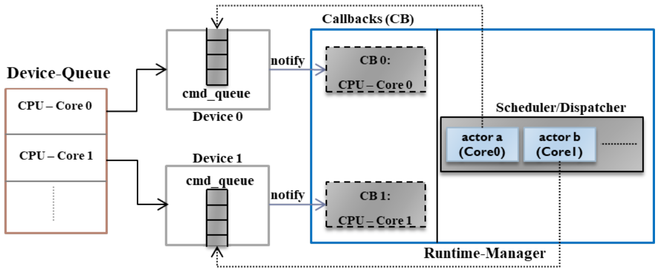


Fig. 4: Actor's invocation-mechanism.

object for each actor that provides the desired information to the host such as, the associated FIFO buffers, the actor's status (idle, running or blocked), the associated kernel, etc. Moreover, the host also creates a *Device-Queue* using the OpenCL specification that lists all the available devices of the target hardware. Each element of this queue provides a command queue of a device, where the actors can be mapped for execution as shown in Fig. 4. 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).

The *Runtime-Manager*, as shown in Fig. 1 and Fig. 4, is a part of the host that exploits different components and provides: the schedulers for scheduling actors based on the proposed SDF and the DDF MoCs, the communication mechanism between the host and kernels, a dispatcher for mapping actors to devices, and the status update mechanism using specialized callbacks.

Scheduling and dispatching. As the main vision of the proposed design flow is to support the heterogeneous combination of particular DPNs, we developed schedulers for the used classes of DPNs using a common dynamic round robin scheduling scheme. Based on the used MoC, the Runtime-Manager invokes the corresponding scheduler that iterates through the Actors-Queue in a round robin fashion, and looks for an actor that is ready for execution. To this end, the scheduler based on the proposed SDF MoC examines each actor based on the semantics explained in Section III-B2. Similarly, the scheduler based on the proposed DDF MoC tests each actor for execution based on the semantics discussed in Section III-B3. Following the underlying semantics, the invoked scheduler fetches a ready actor from the list. Regardless of which scheduler is evoked, the Runtime-Manager then examines the Device-Queue and finds the device with the least weight (i.e., a device assigned with least number of actors), and dispatches the fetched actor on that device. The generated kernel of the dispatched actor is then executed based on the used MoC. As this paper does not focus on presenting efficient mapping of executions on devices, therefore, a simple weighted dispatching scheme is employed.

Communication and the status update mechanism. The communication between the host (FIFO buffers) and kernels is realized using OpenCL buffers. For each bounded FIFO buffer, an OpenCL buffer is created with the same structure and size of the FIFO buffer. Moreover, a status update mechanism is

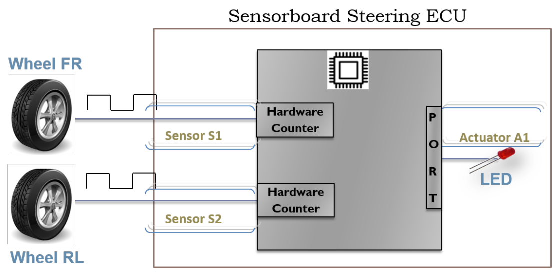


Fig. 5: The hardware design of the Speedometer.

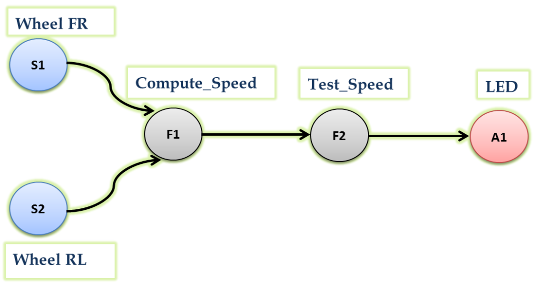


Fig. 6: The dataflow model for the Speedometer.

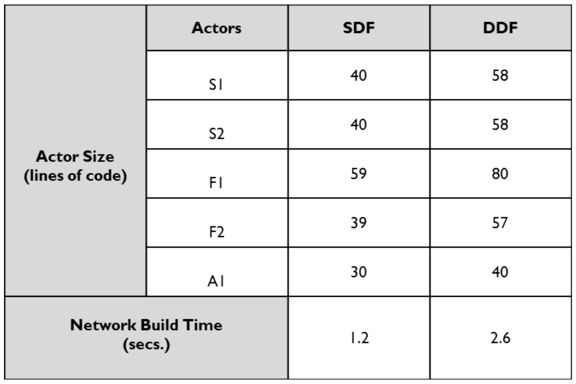
developed using callbacks as shown in Fig. 4. The Runtime-Manager generates a callback interface each, for every existing device in the Device-Queue. The scheduler sets up a callback event for each fetched actor and links it with the callback handler of the device where it is dispatched. Hence, the completion of the kernel of the dispatched actor automatically notifies the Runtime-Manager by invoking the callback handler of the used device. The callback handler performs a set of tasks including: retrieving data from the kernel (OpenCL buffers), updating all the FIFO buffers of the actor, updating the Actors-Queue as well as device's load, updating the OpenCL buffers and so on. The FIFO buffers are updated differently for the proposed SDF and the DDF MoC. Based on the SDF MoC, the data rate of an actor remains fixed in each execution, and therefore each FIFO buffer is simply updated based on the specified static data rate. On the contrary, based on the DDF MoC, the data rate of an actor can change per execution. Therefore, the data rate of each FIFO buffer per execution is first computed and finally each FIFO buffer of the actor is updated accordingly.

IV. Test Case: The ConceptCar

The *ConceptCar* is an experimental vehicle with the objective of testing and verifying modern future car features by deploying different classes of applications. The ConceptCar currently has 8 different ECUs, where each ECU is responsible for a specific operation.

A. Test Application: Dataflow Emulation of the Speedometer

To validate the ability of the framework to produce implementations based on employed MoCs, we present a preliminary test case, namely the dataflow emulation of the *Speedometer*. The Speedometer is an application originally developed for one of the ECUs of the ConceptCar, namely the sensor board steering ECU. The basic hardware design of this application is shown in Fig. 5, where the main idea is to retrieve the frequency of pulses from the sensors (photo-transistors) attached to the front-right (FR) and rear-left (RL) wheel, and to compute the current speed accordingly. The Speedometer accounts for the over-speeding by turning the LED indicator on. The dataflow model for emulating the Speedometer behavior is shown in Fig. 6. The actors *S1* and *S2* provide sampled sensor data, measured by the hardware counters of the sensor board ECU, and collected through the centralized CAN bus of the ConceptCar. Based on this data, the actor *F1* computes the current speed by using the algorithm developed for the sensor board ECU. Next, the measured speed (in meters per second) is tested by the actor *F2* against a threshold value for identifying over-speeding. Finally, the actor *A1* displays the over-speeding status based on the data values provided by *F2*. Each modeled behavior of the Speedometer based on the individual class, i.e., the SDF and the DDF, is then synthesized to the corresponding implementation. The generated implementations are executed on the OpenCL-abstracted target hardware and the experimental results are collected.



	Actors	SDF	DDF
Actor Size (lines of code)	S1	40	58
	S2	40	58
	F1	59	80
	F2	39	57
	A1	30	40
Network Build Time (secs.)		1.2	2.6

Fig. 7: Code size and network build time.

B. Experimental Results

As discussed, dataflow behaviors of the Speedometer are synthesized by the proposed framework to different implementations based on the proposed SDF and the DDF MoC. This preliminary test case thus allows us to observe and analyze the generated implementations for the resulting code size, the total network build time, and the total execution time taken by the complete network for the specific sample sizes.

To this end, the generated code size of each dataflow actor and the total build time for the complete network is depicted in Fig. 7. In contrast to the proposed SDF MoC, where the data rate of each actor per execution is specified statically at compile time, the proposed DDF MoC offers a more flexible semantics, where the decisions on whether to execute actions and consume/produce data are taken dynamically at runtime. Consequently, the latter one accommodates additional code for writing the consumption/production status of data tokens

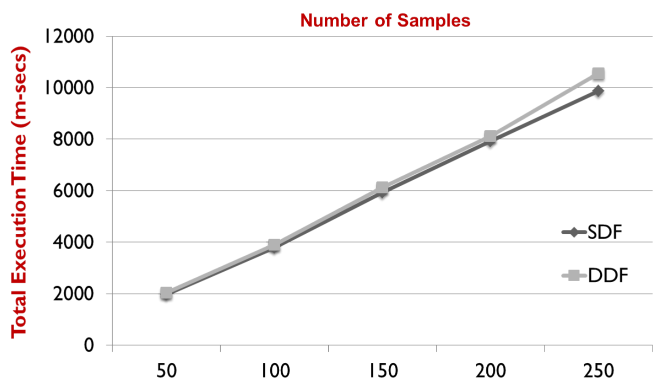


Fig. 8: SDF vs DDF.

for each execution of an actor at runtime. This overhead can therefore be observed from the number of lines of the generated code for each actor and the total network build time for each implementation, as shown in Fig. 7. The generated code based on the DDF MoC for the complete network is approximately 40% greater than based on the SDF MoC, resulting in an additional build time overhead of more than 115%.

Moreover, to analyze and to compare the performance of the proposed MoCs of the framework, the total execution time (in milliseconds) for the complete network is measured against the number of samples (sensor data), as shown in Fig. 8. Based on that, 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 the number of samples increases, this effect induced by the overhead can be clearly observed as shown in Fig. 8. The proposed DDF MoC although offers semantics to model more flexible and data dependent behaviors, but at the cost of the additional runtime overhead. Therefore, it exhibits a trade-off between flexibility and overall performance.

V. Conclusions and Future Work

In this paper, we have presented an automatic model-based design flow to provide synthesis of dataflow behaviors based on the synchronous dataflow (SDF) and the dynamic dataflow (DDF) MoC. The complete design flow including modeling, synthesis and the execution has been explained in detail. To this end, an abstract notion of a dataflow network is introduced which then used differently to formally explain the proposed classes of DPNs. An abstract notion of an actor is used to describe in detail the code generators of the considered MoCs. The complete synthesis scheme including the runtime manager is explained, where all the low-level implementation details are presented. We demonstrated the proposed synthesis design flow by a preliminary test case based on an automotive research platform. The experimental results are carried out based on the code size, the network build time and the total execution time for each implementation.

Overall, the presented work emphasized on the synthesis of individual classes of DPNs. Future work will support the synthesis of heterogeneous DPNs, i.e., where a combination of different DPN actors is allowed. Also, we plan to extend the framework with further classes of DPNs.

References

[1] J. Dennis, “First version of a data-flow procedure language,” in *Programming Symposium*, ser. LNCS, B. Robinet, Ed., vol. 19. France: Springer, 1974, pp. 362–376.

[2] G. Kahn and D. MacQueen, “Coroutines and networks of parallel processes,” in *Information Processing*, B. Gilchrist, Ed. North-Holland, 1977, pp. 993–998.

[3] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete, “Cyclo-static dataflow,” in *International Conference on Acoustics, Speech and Signal Processing*. Michigan, USA: IEEE Computer Society, 1995, pp. 3255–3258.

[4] E. Lee and D. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[5] J. Buck, “Scheduling dynamic dataflow graphs with bounded memory using the token flow model,” Ph.D. dissertation, University of California, California, USA, 1993, phD.

[6] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming heterogeneity – the Ptolemy approach,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, January 2003.

[7] T. Kuhn, T. Forster, T. Braun, and R. Gotzhein, “FERAL - framework for simulator coupling on requirements and architecture level,” in *Formal Methods and Models for Codesign*. Portland, USA: IEEE Computer Society, 2013, pp. 11–22.

[8] J. Stone, D. Gohara, and G. Shi, “OpenCL: A parallel programming standard for heterogeneous computing systems,” *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–73, May-June 2010.

[9] J. Eker and J. Janneck, “CAL language report,” EECS Department, University of California at Berkeley, Berkeley, California, USA, ERL Technical Memo UCB/ERL M03/48, December 2003.

[10] O. Rafique and K. Schneider, “A model-based synthesis framework for the execution of dynamic dataflow actors,” in *Intern. Conference on Internet of Things Embedded Systems and Communications*. Hammamet, Tunisia: IEEE Computer Society, 2018.

[11] G. Kahn, “The semantics of a simple language for parallel programming,” in *Information Processing*, J. Rosenfeld, Ed. Stockholm, Sweden: North-Holland, 1974, pp. 471–475.

[12] E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli, “High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms,” *Journal of Real-Time Image Processing*, vol. 9, no. 1, pp. 251–262, 2014.

[13] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, “ORCC: multimedia development made easy,” in *Intern. conference on Multimedia*. Barcelona, Spain: ACM, 2013, pp. 863–866.

[14] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, “Scenario-based design flow for mapping streaming applications onto on-chip many-core systems,” in *Compilers, Architecture, and Synthesis for Embedded Systems*. Finland: ACM, 2012, pp. 71–80.

[15] L. Schor, A. Tretter, T. Scherer, and L. Thiele, “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.

[16] W. Lund, S. Kanur, J. Ersfolk, L. Tsiopoulos, J. Lilius, J. Haldin, and U. Falk, “Execution of dataflow process networks on OpenCL platforms,” in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Turku, Finland: IEEE Computer Society, 2015, pp. 618–625.

[17] J. Boutellier and I. Hautala, “Executing dynamic data rate actor networks on OpenCL platforms,” in *Signal Processing Systems*. TX, USA: IEEE Computer Society, 2016, pp. 98–103.

[18] J. Boutellier, J. Wu, H. Huttunen, and S. Bhattacharyya, “PRUNE: Dynamic and decidable dataflow for signal processing on heterogeneous platforms,” *IEEE Transactions on Signal Processing*, vol. 66, no. 3, pp. 654–665, February 2018.

[19] C. Lucarz, M. Mattavelli, M. Wipliez, G. Roquier, M. Raulet, J. Janneck, I. Miller, and D. Parlour, “Dataflow/Actor-Oriented language for the design of complex signal processing systems,” in *Conference on Design and Architectures for Signal and Image Processing*, Bruxelles, Belgium, 2008, pp. 1–8.