Technische Universiteit
Eindhoven
University of Technology

Modeling Static-Order Schedules in Synchronous Dataflow Graphs

Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen and Henk Corporaal

This report is an extended version of the following publication. It adds the proofs omitted from the publication. If you want to cite this report, please refer to the paper instead.

M. Damavandpeyma, S. Stuijk, T. Basten, M.C.W. Geilen and H. Corporaal, “Modeling Static-Order Schedules in Synchronous Dataflow Graphs”. In Design, Automation and Test in Europe, DATE 12, Proceedings. Dresden, Germany, 12-16 March, 2012. EDAA, 2012.

ES Reports
ISSN 1574-9517

ESR-2012-01
12 March 2012

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Modeling Static-Order Schedules in Synchronous Dataflow Graphs

Morteza Damavandpeyma[1], Sander Stuijk[1], Twan Basten[1,2], Marc Geilen[1] and Henk Corporaal[1]
[1]Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
[2]Embedded Systems Institute, Eindhoven, The Netherlands
{m.damavandpeyma, s.stuijk, a.a.basten, m.c.w.geilen, h.corporaal}@tue.nl

***Abstract*—Synchronous dataflow graphs (SDFGs) are used extensively to model streaming applications. An SDFG can be extended with scheduling decisions, allowing SDFG analysis to obtain properties like throughput or buffer sizes for the scheduled graphs. Analysis times depend strongly on the size of the SDFG. SDFGs can be statically scheduled using static-order schedules. The only generally applicable technique to model a static-order schedule in an SDFG is to convert it to a homogeneous SDFG (HSDFG). This conversion may lead to an exponential increase in the size of the graph and to sub-optimal analysis results (e.g., for buffer sizes in multi-processors). We present a technique to model periodic static-order schedules directly in an SDFG. Experiments show that our technique produces more compact graphs compared to the technique that relies on a conversion to an HSDFG. This results in reduced analysis times for performance properties and tighter resource requirements.**

I. Introduction

Synchronous dataflow graphs (SDFGs) are widely used to model digital signal processing and multimedia applications [1]–[4]. Model-based design-flows (e.g., [1], [5]–[8]) model binding and scheduling decisions into the SDFG. This enables analysis of performance properties (e.g, throughput [9]) or resource requirements (e.g., buffer sizes [10]) under resource constraints. Fig. 1 shows an example of an SDFG with four *actors* and three *channels*. An essential property of SDFGs is that every time an actor *fires* (executes) it consumes the same amount of tokens from its input edges and produces the same amount of tokens on its output edges. These amounts are called the *rates* (indicated next to the channel ends when the rates are larger than 1). The fixed port rates make it possible to statically schedule SDFGs.

Many SDFG analysis algorithms, e.g., throughput calculation or buffer sizing, are straightforward when a single processor platform is used. For instance, the buffer sizes can be determined by executing the SDFG according to a given schedule. However, in a multi-processor environment, SDFG analysis algorithms are not trivial because of the inter-processor communication amongst other reasons. For a multi-processor, it is possible to construct per processor to which actors of the SDFG are bound, a finite schedule that sequentially orders the actor firings and which is repeated indefinitely. Such a schedule is called a *periodic static-order schedule* (*PSOS*). PSOSs specify the order of actor firing which separates them from fully static schedules, which determine absolute start times of actors. A model-based design-flow usually uses PSOSs for an application modeled with an SDFG. In this way timing (throughput) and memory usage (buffers) can be analyzed.

This work was supported in part by the Dutch Technology Foundation STW, project NEST 10346.

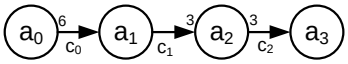


Fig. 1. An example SDFG

There is only one technique [11] known to model PSOSs in an SDFG. This technique requires a conversion of an SDFG to a so-called *homogeneous SDFG* (HSDFG) in which all rates are equal to one [2]. Fig. 2 (without the blue edges) shows the equivalent HSDFG of our example SDFG of Fig. 1. The PSOS modeling technique of [11] sequentializes the actor firings by inserting a channel between each pair of consecutive actors in a processor schedule. At the end of each schedule, it adds a channel with one initial token from the last to the first actor in the schedule. All of these ensure an indefinite execution of the graph according to the schedules. The technique from [11] adds in total 15 channels to the HSDFG of our example graph (the blue edges in Fig. 2) to model PSOSs $s_0 = \langle a_0(a_2)^2\rangle^*$ and $s_1 = \langle (a_1)^5(a_3)^3 a_1(a_3)^3\rangle^*$.

The SDFG to HSDFG conversion can lead to an exponential increase in the size of the graph. For example, converting the SDFG of an H.263 decoder [10] to the equivalent HSDFG increases the graph size from 4 actors to 200 actors. The run-time of SDFG analysis algorithms depends amongst others on the size of the graph. As a result, the run-time of many SDFG analysis algorithms may increase drastically when modeling PSOSs in the graph using the technique from [11]. For example, the buffer sizing algorithm from [10] takes less than 1 ms on the SDFG of an H.263 decoder. Modeling a schedule into this SDFG using the technique from [11], the run-time of the same algorithm increases to 1330 ms. SDFG analysis algorithms are usually repeated more than once in an iterative design-flow. For example, the design-flow from [6] performs 8 throughput calculations to determine the right solution for an H.263 decoder. Hence, it is vital to keep the size of the schedule-extended graph as small as possible to provide a fast and practical design trajectory. There is a second drawback to the technique from [11]. The original graph structure is

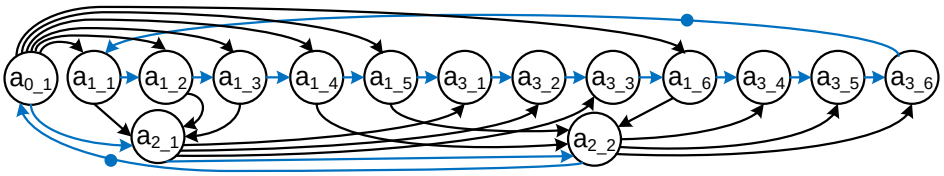


Fig. 2. PSOSs s_0 and s_1 modeled in the SDFG of Fig. 1 using the technique from [11].

lost due to the conversion to an HSDFG. A single channel in an SDFG corresponds to a set of channels in the HSDFG. As a result, common buffer sizing techniques cannot find the minimal buffer size for the original SDFG. The H.263 decoder buffer sizes are for example overestimated by 43% when applying the technique of [10] to the HSDFG. Note that a conversion to an HSDFG may be required in a code generation step. However, if this conversion can be delayed until all analyses are carried out on the SDFG, then this can save a significant amount of resources (e.g., buffer space) and analysis time.

A novel technique is needed to model any PSOS in an SDFG. This technique should limit the increase in the number of actors such that analysis times do not increase too much when analyzing the SDFG with its schedules. The technique should also preserve the original graph structure as this enables accurate analysis of graph properties such as buffer sizes. This paper presents a technique that satisfies both requirements. The technique can be used in any model-based design-flow that models PSOSs into the SDFG (e.g., [1], [5]–[8]). The proposed technique can also directly be used to model PSOSs in scenario-aware dataflow graphs [12].

The remainder of the paper is structured as follows. The next section discusses related work. Sec. III introduces SDFGs. Sec. IV formalizes SDFG schedules. Sec. V presents our technique to model PSOSs in an SDFG. Sec. VI contains the correctness proof of the presented technique. We evaluate our technique by applying it to several realistic applications in Sec. VII. Sec. VIII concludes.

II. Related work

The technique from [11] is the only available technique to model PSOSs in an SDFG. As already explained, this technique may result in a long run-time for analysis algorithms and/or inaccurate results from these algorithms. Our technique alleviates both shortcomings of the technique from [11]. The work in [13] models the effect of a budget scheduler or preemptive TDMA on the temporal behavior of the SDFG, either by computing an accurate worst-case response time, or more precisely by introducing additional actors into a latency-rate model. In contrast, for non-preemptive schedules, such as PSOSs, we focus on the ordering of actor firings; their execution time remains the same. We enforce an SDFG to follow the PSOSs selected for each processor. This allows SDFG analysis to obtain properties like throughput or buffer sizes for the scheduled SDFG. This is also true for the models of [13]. However, our results are tighter since we do not overestimate response times. Since we only use the basic components of an SDFG (e.g., actors and channels) to model schedules in an SDFG, our schedule-extended SDFG can be directly used in any model-based design-flow (e.g., [1], [5]–[8]). Ref [14] uses some new (custom) components, e.g., $if{-}then{-}else$, to model schedules in an SDFG. The common model-based design-flows do not support these components and it is not possible to model these components by using the basic components of an SDFG. Our technique eliminates the need for any new (custom) component. As a result, any analysis technique for SDFGs is directly applicable on the schedule-extended SDFG.

III. Synchronous Dataflow Graphs

Let $\mathbb{N}$ denote the positive natural numbers, $\mathbb{N}_0$ the natural numbers including 0, and $\mathbb{N}_0^\infty$ the natural numbers including 0 and infinity (∞). Formally we define an SDFG as follows. We assume a set $Ports$ of ports, and with each port $p \in Ports$ we associate a finite rate $Rate(p) \in \mathbb{N}$.

Definition 1. (Actor) *An actor a_i is a tuple (In, Out) consisting of a set $In \subseteq Ports$ of input ports (denoted by $In(a_i)$) and a set $Out \subseteq Ports$ of output ports (denoted by $Out(a_i)$) with $In \bigcap Out = \emptyset$.*

Definition 2. (SDFG) *An SDFG is a tuple (A, C) consisting of a finite set A of actors and a finite set $C \subseteq Ports^2$ of channels. The channel source is an output port of some actor, the destination is an input port of some actor. All ports of all actors are connected to precisely one channel, and all channels are connected to ports of some actor. For every actor $a_i = (In, Out) \in A$, we denote the set of all channels that are connected to the ports in In (Out) by $InC(a_i)$ ($OutC(a_i)$).*

Fig. 1 shows an example of an SDFG with four *actors* ($A = \{a_0, a_1, a_2, a_3\}$) and three *channels* ($C = \{c_0, c_1, c_2\}$). These actors communicate with *tokens* sent from one actor to another over the channels. Channels may contain tokens, depicted with a solid dot (and an attached number in case of multiple tokens). An essential property of SDFGs is that every time an actor *fires* (executes) it consumes the same amount of tokens from its input edges and produces the same amount of tokens on its output edges. These amounts are called the *rates* (indicated next to the channel ends when the rates are larger than 1). An actor can only fire if sufficient tokens are available on the edges from which it consumes. Tokens thus capture dependencies between actor firings. Such dependencies may originate from data dependencies, but also

from for example dependencies on shared resources. When an actor a_i starts its firing, it removes $Rate(q)$ tokens from all $c(p,q) \in InC(a_i)$ and when it ends, it produces $Rate(p)$ tokens on every $c(p,q) \in OutC(a_i)$. The rates determine how often actors have to fire with respect to each other such that the distribution of tokens over all channels is not changed. This property is captured in the repetition vector of an SDFG.

Definition 3. (REPETITION VECTOR) *A repetition vector of an SDFG (A, C) is a function $\gamma : A \to \mathbb{N}_0$ such that for every channel $c(p,q) \in C$ from $a_i \in A$ to $a_j \in A$, $Rate(p) \cdot \gamma(a_i) = Rate(q) \cdot \gamma(a_j)$. A repetition vector γ is called non-trivial if and only if for all $a_i \in A, \gamma(a_i) > 0$. An SDFG is called consistent if and only if it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector, which is designated as the repetition vector of the SDFG.*

The repetition vector of the SDFG shown in Fig. 1 is equal to $(a_0, a_1, a_2, a_3) \to (1, 6, 2, 6)$. This shows that the SDFG is consistent as its repetition vector is non-trivial. Consistency and absence of deadlock are two important properties for SDFGs which can be verified efficiently [15], [16]. Any SDFG which is not consistent requires unbounded memory to execute or it eventually deadlocks. When an SDFG deadlocks, no actor is able to fire, which is due to an insufficient number of tokens in a cycle of the graph. Any SDFG which is inconsistent or deadlocks is not useful in practice. Therefore, we limit ourself to consistent and deadlock-free SDFGs.

There exists a special class of SDFGs in which all port rates are equal to 1. These graphs are called *homogeneous SDFGs* (HSDFGs) [16]. Any consistent SDFG can be converted to an HSDFG [2], [16] that is equivalent from the timing perspective. This conversion may however lead to an exponential increase in the number of actors and it has an impact on the speed and possibly also the accuracy of the result of analysis techniques.

IV. SDFG STATIC-ORDER SCHEDULING

A firing of an actor leads to the consumption of tokens from its input channels and the production of tokens on its output channels. In order to capture the behavior of an SDFG, we need to keep track of the distribution of tokens over the channels. The following concept is defined to measure quantities related to channels (e.g., the number of tokens present in channels).

Definition 4. (SDFG STATE) *A state of an SDFG (A, C) is a function $\omega : C \longrightarrow \mathbb{N}_0$ that returns the number of tokens stored in each channel. Each SDFG has an initial state ω_0 denoting the number of tokens that are initially stored in the channels.*

An actor can only be fired if there are sufficient tokens in all of its input channels. An actor that satisfies this condition in a particular state is said to be enabled in this state.

Definition 5. (ENABLED ACTOR) *An actor $a_i \in A$ is called enabled in a state ω_j of SDFG (A, C) if and only if $\omega_j(c) \geq Rate(q)$ for each channel $c(p,q) \in InC(a_i)$.*

Definition 6. (ACTOR FIRING) *The firing of an actor $a_i \in A$ in an SDFG (A, C) which is in state ω_j results in the transition from state ω_j to state ω_{j+1} and is denoted by $\omega_j \xrightarrow{a_i} \omega_{j+1}$. This transition results in the consumption of $Rate(q)$ tokens from each channel $c(p,q) \in InC(a_i)$ and the production of $Rate(p')$ tokens in each channel $c(p',q') \in OutC(a_i)$, i.e., $\omega_{j+1}(c) = \omega_j(c) - Rate(q) + Rate(p')$.*

Consider again our example graph shown in Fig. 1. Its initial state ω_0 is equal to $(c_0, c_1, c_2) \to (0, 0, 0)$. In this state, actor a_0 is enabled. Firing actor a_0 would result in a transition from state ω_0 to a state $(6, 0, 0)$. We use this concept of states and transitions to formalize the execution of an SDFG.

Definition 7. (EXECUTION) *An execution σ of an SDFG (A, C) is an infinite alternating sequence of states and transitions $\omega_0 \xrightarrow{a_i} \omega_1 \xrightarrow{a_j} \cdots$ starting from some designated initial state ω_0.*

In a multi-processor system multiple actors may be bound to the same processor. These actors may be enabled at the same time. In such a situation, a schedule is needed to determine the order in which these enabled actors are fired on the processor. The fixed port rates make it possible to statically schedule SDFGs with a finite schedule per processor that orders the actor firings for that processor and which is repeated indefinitely. Such a schedule is called a periodic static-order schedule (PSOS). Note that in a multi-processor system, a separate static-order schedule should be constructed for each processor. Each schedule should only include actors bound to this specific processor.

Definition 8. (PERIODIC STATIC-ORDER SCHEDULE (PSOS)) *A periodic static-order schedule is a finite ordered list of (a sub-set of) actors in an SDFG (A, C). A periodic static-order schedule is denoted by $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ where each $\alpha_j|_{1 \leq j \leq n}$ represents an actor from A and $n \in \mathbb{N}$ is the length of the schedule s_i, represented by $n = |s_i|$. The set A_i contains all actors that appear at least once in s_i ($A_i \subseteq A$).*

Definition 9. (SDFG ITERATION) *Assume SDFG (A,C) has repetition vector γ. An SDFG iteration is a set of actor firings such that for each $a_i \in A$, the set contains $\gamma(a_i)$ firings of a_i.*

Definition 10. (PSOS ITERATION) *Given a PSOS $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ that schedules a subset of actors A_i. A PSOS iteration is a sequence of actor firings respecting s_i starting from the actor α_1 and ending with the actor α_n with a length equal to the length of the schedule s_i and including only actor firings from actors in A_i. We use $CNT(a_j, s_i)$ to denote the count, i.e., number of appearances, of actor a_j in one iteration of the PSOS s_i.*

A looped schedule, $s_i = \langle (\alpha_1)^{\beta_1} (\alpha_2)^{\beta_2} \cdots (\alpha_m)^{\beta_m} \rangle^*$, is defined as a successive execution of α_1 repeated β_1 times followed by α_2 repeated β_2 times and so on, where each

$\alpha_j|_{j\in\mathbb{N}}$ is either an actor firing or a (nested) looped schedule and $\beta_j \in \mathbb{N}|_{j\in\mathbb{N}}$.

A looped schedule (LS) is a way to represent schedules in a short format. A schedule in LS format can always be converted to a PSOS that adheres to Def. 8. For compactness, we represent in this paper PSOSs in LS format.

Assume that the SDFG in Fig. 1 is mapped to a platform with two processors. Actors a_0 and a_2 are mapped on the first processor (P_0) with $s_0 = \langle a_0(a_2)^2\rangle^*$ and actors a_1 and a_3 are mapped on the second processor (P_1) with $s_1 = \langle (a_1)^5(a_3)^3 a_1(a_3)^3\rangle^*$.

When a consistent and deadlock-free SDFG is executed according to one or more PSOSs, the channels of the SDFG need bounded memories (according to Theorem 1 from [17]). The resulting execution consists of a finite sequence of states and transitions, called the transient phase (σ_{tr}), followed by a sequence of states and transitions which is repeated indefinitely and is called the periodic phase (σ_{pr}) [9]. The number of actor appearances in the PSOS is a fraction or multiple of its repetition vector entry. Formally, each actor a_i in the PSOS should appear $r \cdot \gamma(a_i)$ times in the PSOS (with $r = \frac{u}{v}$ where $u, v \in \mathbb{N}$) and the value r is identical for all actors in the PSOS [9]. This follows from the SDFG property that firing each actor as often as indicated in the repetition vector results in a token distribution that is equal to the initial token distribution. In the paper, the term *normalized PSOS* is used to refer to a PSOS with r equal to 1.

Definition 11. (NORMALIZED PSOS) *A PSOS s_i is called normalized if and only if each actor $a_j \in A_i$ appears $\gamma(a_j)$ times in one iteration of the PSOS s_i.*

We limit ourselves in the remainder to PSOSs in which r is a *unit fraction* (i.e., $r = \frac{u}{w}$ with $u = 1$ and $w \in \mathbb{N}$), although our technique can also be directly applied to model other PSOSs (i.e., in which $u \in \mathbb{N}$).

The order of actor firings in an execution $\omega_0 \xrightarrow{\alpha_1} \omega_1 \xrightarrow{\alpha_2} \cdots$ is an ordered list $\{\alpha_1, \alpha_2, \cdots\}$ ($\alpha_j|_{j\in\mathbb{N}}$ represents an actor from A). The notation *orderList*(σ, A_i) represents the ordered list extracted from execution σ while actors which do not belong to A_i are omitted from the ordered list. When an SDFG is executed according to a PSOS, we say that the corresponding execution of the SDFG satisfies that PSOS. We use the following definition to formalize this term.

Definition 12. (SATISFACTION) *The execution σ of an SDFG satisfies the PSOS s_i for actors in $A_i \subseteq A$ if and only if for the ordered list* orderList(σ, A_i) *results in the same actor ordering as specified in the PSOS s_i.*

V. MODELING PERIODIC STATIC-ORDER SCHEDULES

In this section, we introduce a technique to model PSOSs in an SDFG. Algorithm 1 encapsulates our technique, called decision state modeling (DSM). Fig. 3 depicts the corresponding SDFG of Fig. 1 which models the PSOSs s_0 and s_1 using DSM. The remainder of this section discusses different parts of the algorithm in detail. There are several reasons why an SDFG cannot model PSOSs naturally. The following subsections discuss them and illustrate how we address them.

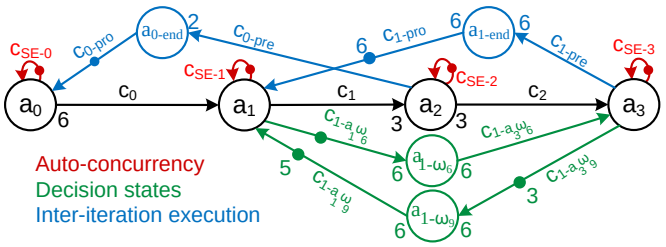


Fig. 3. PSOSs s_0 and s_1 modeled in the SDFG of Fig. 1 using DSM.

The description of some basic functions used in Algorithm 1 is as follows. The function *AA(G, a_{new})* is responsible to include the actor a_{new} in the SDFG G. The function *AC(G, c_{new}, a_{src}, a_{dst}, $srcRate$, $dstRate$, $initTok$)* adds the channel c_{new} from the source actor a_{src} to the destination actor a_{dst}; the production (consumption) rate of a_{src} (a_{dst}) on this channel is equal to $srcRate$ ($dstRate$); this channel is initialized with $initTok$ tokens. The function *BEF(a_k,ω_j,s_i)* (*AFT(a_k,ω_j,s_i)*) returns the number of times that a_k appears before (after) state ω_j in one iteration of the PSOS s_i.

A. Auto-concurrency

An actor $a_i \in A$ in an SDFG state ω_j can possibly be enabled multiple times simultaneously in the state ω_j. This property is called auto-concurrency. The firings related to actor a_i should occur sequentially according to the PSOS to which actor a_i belongs. This sequential execution can be enforced by adding a self-edge with one initial token to actor a_i (Line 1 in Algorithm 1). In Fig. 3, channels $c_{SE-0} - c_{SE-3}$ (shown in red) are used to prevent any auto-concurrency in the SDFG of Fig. 1.

B. Inter-iteration execution

Enabled actors in a PSOS belonging to the next PSOS iteration prevent the execution of the SDFG from following the given PSOS; lines 4-8 in Algorithm 1 are used to control this undesirable actor enabling. This part of the algorithm adds (per PSOS) one actor and two channels to create a dependency between the last and first actor appearing in the PSOS. The added components limit, within one PSOS iteration, the firing of the first actor in the PSOS (i.e., a_F) to the count of actor a_F (i.e., $CNT(a_F, s_i)$) in one iteration of the PSOS s_i. The function $CNT(a_F, s_i)$ in DSM returns the count of the actor a_F in one iteration of the PSOS s_i. The next iteration of the PSOS s_i can only commence if the last actor in PSOS s_i (i.e., a_L) fires $CNT(a_L, s_i)$ times in one iteration of the PSOS s_i. In other words, the next iteration of a PSOS can only commence after the completion of the current iteration of this PSOS. In Fig. 3, actor a_{0-end} and channels c_{0-pre} and c_{0-pro} are added to prevent any inter-iteration execution in PSOS s_0. Actor a_{1-end} and channels c_{1-pre} and c_{1-pro} are added to prevent any inter-iteration execution in schedule s_1. These elements are shown in our example in blue in Fig. 3.

Algorithm 1: Decision State Modeling (DSM)

input : SDFG $G(A, C)$, PSOSs $\{s_0, \cdots, s_n\}$
output: G extended with schedules $\{s_0, \cdots, s_n\}$

1 add a self edge with 1 initial token for each $a \in A$
2 $\{s'_0, \mu_0, \cdots s'_n, \mu_n\} \leftarrow$ **normalize**$(G, \{s_0, \cdots, s_n\})$
3 **for** $i \leftarrow 0$ **to** n **do**
  /* To control inter-iteration execution */
4 $a_L :=$ *last actor in* s_i
5 $a_F :=$ *first actor in* s_i
6 **AA**(G, a_{i-end})
7 **AC**$(G, c_{i-pre}, a_L, a_{i-end}, 1,$ **CNT**$(a_L, s_i), 0)$
8 **AC**$(G, c_{i-pro}, a_{i-end}, a_F,$ **CNT**$(a_F, s_i), 1,$ **CNT**$(a_F, s_i))$
  /* To control decision states */
9 $\Omega \leftarrow$ **getDecisionStates**$(G, s'_i, \{s'_0, \cdots, s'_n\} \setminus s'_i)$
10 $\Omega \leftarrow$ **reduceDecisionStates**(Ω)
11 $\Omega \leftarrow$ **foldDecisionStates**(Ω, μ_i)
12 **foreach** $\omega_j \in \Omega$ **do**
13 **AA**(G, $a_{i-\omega_j}$)
14 **foreach** $a_k \in \Delta_j$ **do**
15 **if** a_k *is the actor of choice* **then**
16 **AC**$(G, c_{i-a_k\omega_j}, a_k, a_{i-\omega_j}, 1,$ **CNT**$(a_k, s_i),$ **AFT**$(a_k, \omega_j, s_i))$
17 **else**
18 **AC**$(G, c_{i-a_k\omega_j}, a_{i-\omega_j}, a_k,$ **CNT**$(a_k, s_i), 1,$ **BEF**$(a_k, \omega_j, s_i))$

C. Decision states

1) Concept: The state space when executing our example SDFG using the PSOSs s_0 and s_1 is visualized in Fig. 4. In this figure, the actors mapped on processor P_0 (P_1) are surrounded by a square (circle). Auto-concurrency and inter-iteration execution are excluded using the constructs introduced in Sec. V-A and Sec. V-B respectively. The periodic behavior of the PSOSs is obvious from the state space. There are some states in which more than one actor is enabled ($\omega_5 - \omega_9$) on one processor. In such a situation, the execution related to those actors can deviate from the specified PSOS. We use the following definition to formalize such a situation.

Definition 13. (DECISION STATE) *Consider the PSOS s_i which schedules actors $A_i \subseteq A$ and an execution σ of an SDFG (A, C) which satisfies PSOS s_i. A state $\omega_j \in \sigma$ is a decision state if and only if multiple actors from A_i are enabled in ω_j.*

The finite set Ω occurring in DSM contains these decision states for the PSOS being considered. The following terminology is used to describe those enabled actors in a decision state.

Definition 14. (OPPONENT ACTOR SET) *Let $\omega_j \in \Omega$ be a decision state within PSOS s_i. The opponent actor set Δ_j of the decision state ω_j is a finite set which contains all actors that are enabled in decision state ω_j and that belong to A_i.*

The finite set Δ_j represents the opponent actors in the decision state $\omega_j \in \Omega$. One of the enabled actors in a decision state ω_j, in line with the given PSOS s_i, should be selected to get fired. The following definition is used to describe such an actor.

Definition 15. (ACTOR OF CHOICE) *Consider the PSOS s_i which schedules actors $A_i \subseteq A$ and the opponent actor set Δ_j of the decision state ω_j in an execution σ of the SDFG $G(A, C)$ which satisfies PSOS s_i. An actor $a_c \in \Delta_j$ is called the actor of choice of the decision state ω_j if and only if the firing of actor a_c in state ω_j is a necessity for the execution σ in order to satisfy the PSOS s_i.*

One member of the set Δ_j is the actor of choice in decision state ω_j; we denote that actor of choice with $a_c \in \Delta_j$.

Lines 9-18 in DSM show how we deal with uncertainty due to decision states. In the algorithm $n + 1$ ($n \in \mathbb{N}_0$) is the number of processors (or input PSOSs). DSM models the given PSOSs one-by-one iteratively. The ordering of PSOSs in DSM does not have any impact on the final outcome. In each iteration of the for-loop in line 3, we enforce the execution of the actors in the current schedule of interest (i.e., schedule s_i) to follow schedule s_i. The next sub-section explains how decision states of the schedule of interest are extracted. For each $\omega_j \in \Omega$ extracted from s_i, DSM adds an actor ($a_{i-\omega_j}$ in line 13) and one channel between the new actor $a_{i-\omega_j}$ and each opponent actor in the set Δ_j (lines 14-18 in Algorithm 1). In our example, these elements are shown in green in Fig. 3. In practice, the elements added in each decision state (e.g., ω_j) postpone the execution of the actors in $\Delta_j \setminus \{a_c\}$ to the state after decision state ω_j. Hence, a_c (i.e., the actor of choice) is the only actor which can be fired in the state ω_j.

2) Decision state identification: Algorithm 2 shows our proposed technique to detect all decision states. In this algorithm, s_c is the PSOS for which we want to determine the decision states. Assume s_c is a PSOS for the actors mapped on processor P_c. Schedules $s_{o1} \cdots s_{on}$ are PSOSs for the other actors of the SDFG mapped on the other processors (with $P_{o1} \cdots P_{on}$ as the other processors). In Algorithm 2, the input schedules are normalized PSOSs. The function $normalize$ (in line 2 of Algorithm 1) normalizes the input PSOSs. The function returns the normalized PSOSs along with their normalization factors. The normalized PSOS s'_x can be achieved by repeating μ_x times the input PSOS s_x (i.e., $s'_x = (s_x)^{\mu_x}$). μ_x is the normalization factor of s_x and can be calculated by dividing the repetition vector entry of an arbitrary actor in s_x by the count of that actor in the PSOS s_x (in our example, μ_0 and μ_1 are equal to 1).

An actor in the schedule of interest s_c could be affected by the execution of an actor in the other schedules as well as another actor in the schedule s_c. Processors can run at different clock rates; these differences and inter-processor dependencies cause variation in the amount of tokens on the *inter-processor channels* originating from the actors mapped on the other processors to the actors mapped on the processor of interest (i.e., P_c). The amount of tokens on the input channels of an actor determines whether an actor is enabled (ready to execute) or not. Our technique can determine any possible actor enabling when executing s_c by considering the maximum amount of tokens on all inter-processor channels. Each iteration of the schedule of interest s_c requires that the

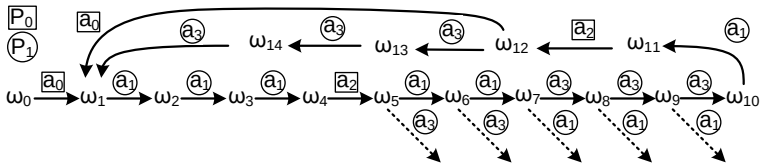


Fig. 4. The state space of the SDFG of Fig. 1 when PSOSs $s_0 = \langle a_0(a_2)^2\rangle^*$ and $s_1 = \langle(a_1)^5(a_3)^3a_1(a_3)^3\rangle^*$ are used.

actors mapped on the other processors are fired up-to at most their repetition vector entry values. Hence, only executing one iteration of the other schedules $s_{o1}\cdots s_{on}$ is enough to provide sufficient tokens on inter-processor channels entering to the actors mapped on processor P_c. Subsequent iterations of the other schedules $s_{o1}\cdots s_{on}$ are possible; this may enable an actor in the schedule s_c to be enabled more than its designated amount in one iteration of the schedule s_c. The inter-iteration prevention constructs introduced in Sec. V-B are used to control this undesired actor enabling. So, we only extract decision states within one iteration of the normalized schedule. Also, DSM does not impose any limitation between PSOSs; PSOSs can independently be iterated if the dependencies in the SDFG allow that. We allow the actors on the other processors to be executed (according to their schedules) as much as they can. The execution of the actors on the other processors will stop at one point either due to their dependency on the actors on the processor P_c or because one iteration of their schedule is completed. The state of the SDFG needs to be preserved to follow the subsequent execution of the actors. This maximal execution of the actors on the other processors is represented by the function *maxExec* in Algorithm 2. After this maximal execution, the amount of tokens on the inter-processor channels entering into the actors on the processor P_c determines any possible enabled actor. The current state (represented by ω_j) will be added to the decision state set (Ω) if more than one actor on the processor P_c is enabled at this state (line 5 in Algorithm 2). All enabled actors will be recorded as opponent actors of the state ω_j (line 6 in Algorithm 2). The execution of the actors on the processor P_c is continued by executing the enabled actor in line with s_c in order to determine all possible decision states (line 7 in Algorithm 2). The function $fire(G, \omega_j, s_c[i])$ fires the actor at the i^{th} position in the PSOS s_c. The maximal execution followed by decision state identification will be iterated to execute one iteration of s_c. In the end, the set Ω contains all possible decision states when executing PSOS s_c. In the SDFG of Fig. 1, five consecutive decision states ($\Omega = \{\omega_5 \cdots \omega_9\}$) exist for PSOS s_1 and no decision state exists for PSOS s_0 (see Fig. 4).

3) Redundant decision states: It is possible to have several consecutive decision states which are postponing the firing of an actor to several states later. For example, three consecutive decision states ($\omega_7 - \omega_9$) exist in Fig. 4 that all postpone the same firing of actor a_1; the added components in decision state ω_7 postpone the sixth firing of a_1 to the state ω_8; the added components in decision state ω_8 postpone the sixth firing of a_1 to the state ω_9; and so on. The latest decision state in the sequence of decision states $\omega_7 - \omega_9$ is enough to postpone the firing of actor a_1 to state ω_{10}. Hence, the decision states $\omega_7 - \omega_8$ are redundant and can be removed from the decision state set Ω. The function *reduceDecisionStates* is responsible for removing redundant decision states. Note that it would be possible to perform this reduction during the decision state identification step. This reduction can remove a significant amount of extra components in the final SDFG. Decision state ω_5 is also redundant according to our optimization. So, only two decision states ω_6 and ω_9 are necessary to model s_1 in the SDFG of Fig. 1.

Algorithm 2: Get Decision States

input : SDFG G, PSOS s_c, PSOSs $\{s_{o1}, \cdots, s_{on}\}$
output: Decision state set Ω

1 $\omega_j \leftarrow$ the initial state of G
2 **for** $i \leftarrow 1$ **to** $|s_c|$ **do**
3 $\omega_j \leftarrow$ **maxExec**(G, ω_j, $\{s_{o1}, \cdots, s_{on}\}$)
4 **if** ***sizeof(enabledActors***$(G, \omega_j, s_c)) > 1$ **then**
5 $\Omega \leftarrow \Omega \cup \{\omega_j\}$
6 $\Delta_j \leftarrow$ **enabledActors**(G, ω_j, s_c)
7 $\omega_j \leftarrow$ **fire**(G, ω_j, $s_c[i]$)

4) Decision state folding: In Algorithm 1, the input PSOSs are normalized to find all decision states. The normalization of PSOSs is required to explore all (sufficient) states of an SDFG. Consider PSOSs $s_2 = \langle a_0\rangle^*$ and $s_3 = \langle a_2\ a_1\rangle^*$ for our second example SDFG in Fig. 5. To obtain normalized PSOSs, μ_2 and μ_3 must be equal to 3 and 4 respectively. This leads to the following normalized PSOSs: $s_2' = \langle(a_0)^3\rangle^*$ and $s_3' = \langle(a_2\ a_1)^4\rangle^*$. Decision state identification for PSOS s_3' results in 5 decision states. $\underbrace{\binom{a_2}{\_}\binom{a_1}{a_2}}_{1^{st}\ 2^{nd}}\underbrace{\binom{a_2}{a_1}\binom{a_1}{a_2}}_{3^{rd}\ 4^{th}}\underbrace{\binom{a_2}{a_1}\binom{a_1}{a_2}}_{5^{th}\ 6^{th}}\underbrace{\binom{a_2}{\_}\binom{a_1}{\_}}_{7^{th}\ 8^{th}}$ shows the corresponding execution of s_3'. In construct $\binom{a_x}{a_y}$, a_x is the enabled actor in line with the PSOS and a_y is the other enabled actor if any at all. In this execution, the 1^{st}, 3^{rd}, 5^{th} and 7^{th} states are similar in behavior. In other words, the actor a_2 should be fired in all of those states.

Modeling a repetitive behavior for a PSOS s_i, also models this behavior for its normalized PSOS (i.e., $s_i' = (s_i)^{\mu_i}$). By considering this fact, we can merge decision states appearing in all μ_i repetitions of the PSOS s_i. We call this optimization *decision state folding* (line 11 in Algorithm 1). Folding groups the similar states. In our example, the 1^{st}, 3^{rd}, 5^{th} and 7^{th} states are grouped and represented with one state. Similar state grouping can be performed for the 2^{nd}, 4^{th}, 6^{th} and 8^{th}

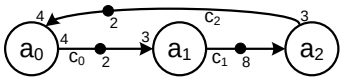
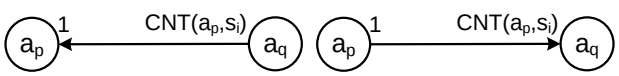


Fig. 5. An example SDFG



(a) a_q is producer (b) a_q is consumer

Fig. 6. Extra actor a_q added by DSM in different situations.

states. So, the above execution shrinks to $\binom{a_2}{a_1}\binom{a_1}{a_2}$. If there is a decision state in any of the similar states in the original execution, a decision state will be placed in the substitution state of those states. In practice, a decision state in a state of the new folded execution will be considered as a decision state for each of the equivalent states in the original execution. This cannot violate the execution according to the input PSOS because DSM only ensures the execution of the actor of choice in a decision state. This optimization could reduce the number of decision states up to μ_i times in a normalized PSOS s'_i. The decision state in the last state is ignored thanks to our inter-iteration execution prevention (which is explained in Sec. V-B). In our second example, decision state folding reduces the number of decision states from 5 to 1 for s_3.

5) Enforcing a schedule in decision states: In our first example SDFG, only two actors are enabled in decision state ω_6 (i.e., $\Delta_6 = \{a_1, a_3\}$) (see Fig. 4). Actor a_1 is the actor of choice in decision state ω_6 and actor a_3 is the only opponent actor whose execution should be postponed to the state after state ω_6. DSM adds actor $a_{1-\omega_6}$ and channels $c_{1-a_1\omega_6}$ and $c_{1-a_3\omega_6}$ to the graph to control the actor firings in decision state ω_6. DSM also adds actor $a_{1-\omega_9}$ and channels $c_{1-a_1\omega_9}$ and $c_{1-a_3\omega_9}$ to the graph for the other decision state ω_9.

The actor $a_{1-\omega_9}$ is added to enforce the firing of a_3 in decision state ω_9 and to postpone the execution of a_1 to the subsequent state. The actor $a_{1-\omega_9}$ is only responsible for decision state ω_9 and it fires only once in an iteration. This means that its value in the repetition vector of the new graph (i.e., Fig. 3) is one. The production and consumption rates of the ports of the actor $a_{1-\omega_9}$ should be set to a value that preserves the consistency of the SDFG; for this purpose, the port rates of actor $a_{1-\omega_9}$ on its channels (i.e., $c_{1-a_1\omega_9}$ and $c_{1-a_3\omega_9}$) are set to 6. The added dependency channels from the newly added actor in decision state ω_j (e.g., $a_{1-\omega_9}$ in decision state ω_9) to the opponent actors which are not the actor of choice (e.g., a_1 in decision state ω_9) only provide enough tokens for their execution in states $\omega_0 - \omega_{j-1}$ (e.g., 5 tokens for a_1 in states $\omega_0 - \omega_8$); these actors cannot be enabled due to the lack of tokens in the newly added channels in the corresponding decision state (e.g., there will be no token in channel $c_{1-a_1\omega_9}$ in decision state ω_9). Hence only the actor of choice amongst the opponent actors of a decision state will be enabled in that state (e.g., only a_3 can fire in decision state ω_9). The firing of the postponed actors in a decision state (e.g., decision state ω_j) will not depend on the newly added actor in the decision state (i.e., $a_{i-\omega_j}$) after firing of the actor of choice in ω_j. For example, there will be 6 tokens in channel $c_{1-a_3\omega_9}$ after the firing of actor a_3 (i.e., the actor of choice) in decision state ω_9; hence, the actor $a_{1-\omega_9}$ can immediately fire and its execution will provide sufficient tokens for later firings of actor a_1. So, the postponed actor in decision state ω_9 will not be dependent on actor $a_{1-\omega_9}$ for its later execution in the current iteration of the PSOS s_1.

The firing of actor a_3 after decision state ω_9 produces 3 tokens in channel $c_{1-a_3\omega_9}$ and the firing of actor a_1 after decision state ω_9 consumes 1 token from channel $c_{1-a_3\omega_9}$; as a result, the amount of tokens in the new channels are reset to the initial values at the end of one iteration of the schedule s_1. Hence, the periodic behavior is also achievable for the added components. The components added in decision state ω_6 show similar behavior as the components added in decision state ω_9.

VI. CORRECTNESS OF DSM

This section discusses the correctness of DSM in modeling a single PSOS for a subset of the actors of the SDFG. If we can model a single PSOS in the SDFG, then we can also model multiple PSOSs by simply applying the algorithm multiple times.

DSM adds some extra actors and channels to model the PSOS s_i in SDFG $G(A, C)$. The sets A_{s_i} and C_{s_i} represent those extra actors and channels respectively. The notation $G'(A', C')$ represents the SDFG which models the PSOS s_i in the SDFG G using DSM where $A' = A \cup A_{s_i}$ and $C' = C \cup C_{s_i}$. The following proposition shows the consistency of the schedule-extended SDFG G'.

Proposition 1. *The SDFG $G'(A', C')$ which models PSOS s_i in the consistent SDFG $G(A, C)$ is consistent.*

Proof. The SDFG $G(A, C)$ is consistent. In other words, a non-trivial repetition vector γ exists for SDFG G. DSM adds some extra actors A_{s_i} and channels C_{s_i} to the SDFG G in order to model PSOS s_i. We need to show that a non-trivial repetition vector γ' exists for SDFG G'. The repetition vector equality related to each self-loop $c(a_p, a_p) \in C_{s_i}, a_p \in A_i$ added by DSM (in line 1 of Algorithm 1) to remove auto-concurrency is always valid because the source and destination actor of the self-loop channel are identical with production and consumption rate equal to one. The rates of the other channels added by DSM (for decision states or inter-iteration execution) share the following properties: (1) the newly added channel $c \in C_{s_i}$, which is added by DSM (in all lines 7, 8, 16 or 18 of Algorithm 1), is between an actor $a_p \in A(= A' \setminus A_{s_i})$ and an actor $a_q \in A_{s_i}$; (2) the rate of the new channel on the side of the actor a_q is equal to the count of the actor a_p in one iteration of PSOS s_i (i.e., $CNT(a_p, s_i)$); (3) the rate of the new channel on the side of the actor a_p is equal to one. From these properties we conclude that an actor $a_q \in A_{s_i}$, which is added by DSM (in both line 6 and 13 of Algorithm 1), fires only once in each iteration of the PSOS s_i; if the actor a_q is

the producer (see Fig. 6(a)) of the newly added channel (i.e., $(a_q, a_p) \in C_{s_i}$), the only firing of actor a_q in one iteration of the PSOS s_i provides $CNT(a_p, s_i)$ tokens for all firings of actor a_p in one iteration of the PSOS s_i and if the actor a_q is the consumer (see Fig. 6(b)) of the newly added channel (i.e., $(a_p, a_q) \in C_{s_i}$), all firings of actor a_p in one iteration of the PSOS s_i provide $CNT(a_p, s_i)$ tokens for only one firing of actor a_q.

Let set A_i be the set of the actors in the PSOS s_i. Consider that each actor $a_p \in A_i$ appears $r \cdot \gamma(a_p)$ times in the PSOS s_i ($r = \frac{u}{v}$ where $u, v \in \mathbb{N}$) where the value r is identical for all actors in the PSOS s_i. The appearance count of the actor $a_p \in A_i$ in the PSOS s_i is represented by $CNT(a_p, s_i)$ and it is assumed to be equal to $\frac{u}{v} \cdot \gamma(a_p)$ where $u, v \in \mathbb{N}$. We can write the above statement as follow: $CNT(a_p, s_i) \cdot v = \gamma(a_p) \cdot u$. From this equation we can conclude that u iterations of the SDFG G cause v iterations of the PSOS s_i, leading to v firings of each of the actors added by DSM (i.e., actors from the set A_{s_i}). So, in u iterations of the SDFG G (or v iterations of the PSOS s_i) the following equation holds for each channel $(a_p, a_q) \in C_{s_i}$ or $(a_q, a_p) \in C_{s_i}$ (where $a_p \in A_i$ and $a_q \in A_{s_i}$):

$$\underbrace{CNT(a_p, s_i)}_{Rate(a_q)} \cdot \underbrace{v}_{\gamma'(a_q)} = \underbrace{1}_{Rate(a_p)} \cdot \underbrace{\gamma(a_p) \cdot u}_{\gamma'(a_p)} \quad (1)$$

Eqn. 1 shows the existence of a non-trivial repetition vector γ' for the schedule-extended SDFG $G'(A', C')$. $\gamma'(a_q)$ (where actor $a_q \in A_{s_i}$) is equal to v and $\gamma'(a_p)$ (where actor $a_p \in A$) is equal to $\gamma(a_p) \cdot u$. □

As stated before, inter-iteration execution could cause an uncertainty in the SDFG execution according to the designated schedule. Proposition 2 states that PSOS inter-iteration execution is eliminated from the schedule-extended SDFG G'. As a result, no SDFG inter-iteration execution can happen in the schedule-extended SDFG G' because an SDFG iteration may contain one or more PSOS iterations.

Proposition 2. *PSOS inter-iteration execution is impossible for any actor appearing in PSOS $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ in the SDFG $G'(A', C')$, which models PSOS s_i in the SDFG $G(A, C)$ using DSM.*

Proof. Let set A_i be the set of the actors in the PSOS s_i. In one iteration of a PSOS s_i, an actor $a_p \in A_i$ could be enabled more often than its designated amount (i.e., $CNT(a_p, s_i)$ times). DSM prevents this by creating a dependency from the last actor appearing in PSOS s_i (i.e., actor $a_L = \alpha_n$) to the first actor appearing in PSOS s_i (i.e., actor $a_F = \alpha_1$) (see lines 4-8 in DSM). This dependency is created in the graph by inserting a new actor a_{i-end} and two channels c_{i-pre} and c_{i-pro}. The source (destination) actor of the channel c_{i-pre} is a_L (a_{i-end}) with rate 1 ($CNT(a_L, s_i)$). So one firing of actor a_{i-end} needs $CNT(a_L, s_i)$ tokens available in channel c_{i-pre}; for this purpose actor a_L should fire $CNT(a_L, s_i)$ times. The destination (source) actor of the channel c_{i-pro} is a_F (a_{i-end}) with rate 1 ($CNT(a_F, s_i)$). So the firing of actor a_F related to one iteration of PSOS s_i needs $CNT(a_F, s_i)$ tokens available in channel c_{i-pro}; for this purpose actor a_{i-end} should fire once. The $CNT(a_F, s_i)$ initial tokens in channel c_{i-pro} provides sufficient tokens for $CNT(a_F, s_i)$ times a firing of actor a_F related to one iteration of PSOS s_i. The subsequent firing of actor a_F depends on the firing of the actor a_{i-end} and the firing of the actor a_{i-end} demands $CNT(a_L, s_i)$ times firing of actor a_L. Hence, the firing of actor a_F belonging to the subsequent iteration of s_i can be performed only after actor a_L finishes all of its firings belonging to the current iteration of s_i. In other words, the firing of actor a_F belonging to the subsequent iteration of s_i can only be performed after completion of the current iteration of s_i. The next iteration of the PSOS s_i can only start after completion of the current iteration of the PSOS s_i because actor a_F is the first actor which should be fired in an iteration of the PSOS s_i and other actors in s_i cannot get enabled before the first firing of the actor a_F. This second fact is guaranteed by adding decision state constructs (i.e., lines 12-18 in DSM) for any possible decision state and Proposition 6 below. □

Even after eliminating inter-iteration execution from the SDFG, multiple actors from a schedule may be enabled in an SDFG iteration. In this paper such a state is called a decision state. The following proposition explains that analyzing only one SDFG iteration is enough in order to identify all possible decision states.

Proposition 3. *Executing an SDFG $G(A, C)$ for one iteration is sufficient to determine all possible decision states within a PSOS s_i.*

Proof. Let set A_i be the set of the actors in the PSOS s_i and $A_o = A \setminus A_i$ the remaining actors in A. Consider inter-processor channels $C_{ipc} = \{(a_p, a_q) \in C | a_p \in A_o \wedge a_q \in A_i\}$. The execution of an actor $a_p \in A_o$ where $(a_p, a_q) \in C_{ipc}$ up-to its entry in the repetition vector of the SDFG produces $\gamma(a_p) \cdot Rate(a_p)$ tokens in the corresponding channel $(a_p, a_q) \in C_{ipc}$. Actor $a_q \in A_i$ consumes those produced tokens within one iteration of the normalized PSOS s_i (because $\gamma(a_p) \cdot Rate(a_p) = \gamma(a_q) \cdot Rate(a_q)$). Hence, actors in A_i can receive the required tokens from all inter-processor channels C_{ipc} for one iteration of the normalized PSOS s_i. This means that all possible decision states related to PSOS s_i are detectable.

Actors in A_o could possibly fire more than the amount mentioned above (i.e., corresponding value in vector γ) if the channel dependencies in the SDFG allow additional firings of these actors. This could cause more than enough tokens (for one iteration of the normalized PSOS s_i) in channels C_{ipc}. This could enable an actor in A_i more than its designated amount in one iteration of the normalized PSOS s_i. To avoid this undesirable actor enabling, the inter-iteration execution prevention constructs are used (see Proposition 2). As a result, extra tokens produced by further firing of the actors A_o cannot enable any actor in A_i more than its designated value in one iteration of the normalized PSOS s_i. So, executing actors in

the SDFG up-to their repetition vector entry (i.e., one SDFG iteration) is enough to determine all possible decision states within the PSOS s_i. □

The identified decision states may be redundant. Proposition 4 discuses the proposed decision state reduction in the DSM.

Proposition 4. *Let σ be an execution for an SDFG (A, C) and a PSOS s_i which schedules actors $A_i \subseteq A$. In the execution σ, consider y consecutive decision states $\omega_{x+1}, \omega_{x+2}, \cdots, \omega_{x+y}$. Assume that $a_o \in A_i$ is an opponent actor in each of these decision states but not the actor of choice in any of them. It is sufficient to only consider the last decision state ω_{x+y} to postpone the firing of the opponent actor a_o in those consecutive decision states to the state $\omega_{x+y+1} \in \sigma$.*

Proof. The purpose of the components added by DSM (i.e., lines 13-18 in Algorithm 1) in a decision state $\omega_j \in \Omega$ is to prevent any opponent actor a_o which is not the actor of choice in decision state ω_j from getting enabled in that state and as such to postpone that firing to the state ω_{j+1}. It is assumed that the opponent actor a_o is enabled in the consecutive decision states $\omega_{x+1}, \omega_{x+2}, \cdots, \omega_{x+y}$. Suppose that the opponent actor a_o was fired e times before the first decision state (i.e., ω_{x+1}) where $0 \leq e < \gamma(a_o)$. The actor a_o cannot be enabled in decision sates $\omega_{x+1}, \omega_{x+2}, \cdots, \omega_{x+y-1}$ when actor a_o was fired e times before the decision state ω_{x+1} because the components added by DSM in the last decision state ω_{x+y} prevent the opponent actor a_o from getting enabled for the $(e+1)^{th}$ time in decision state ω_{x+y}. As a result, the opponent actor a_o can also not be enabled in states $\omega_{x+1}, \omega_{x+2}, \cdots, \omega_{x+y-1}$ after adding DSM components for decision state ω_{x+y}. So, the components added by DSM in the last decision state of consecutive decision states are enough to prevent the firing of an opponent actor which is not the actor of choice in those consecutive decision states. □

Decision state folding overlaps the consecutive repetitions of the designated PSOS in an SDFG iteration to reduce the number of decision states. The following proposition states that decision states folding does not dismiss any decision state.

Proposition 5. *Consider PSOS $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ for the subset of actors A_i from SDFG (A, C); assume s_i is repeated μ_i times to form the corresponding normalized PSOS ($s'_i = \langle (s_i)^{\mu_i} \rangle^*$) to identify decision states related to PSOS s_i. After decision state folding all decision states are preserved.*

Proof. Normalization can be done by repeating PSOS s_i μ_i times (μ_i is the normalization factor of s_i). Decision state identification is applied on the normalized PSOS $s'_i = \langle (s_i)^{\mu_i} \rangle^* = \langle \underbrace{\alpha_1 \alpha_2 \dots \alpha_n}_{1^{st}} \underbrace{\alpha_1 \alpha_2 \dots \alpha_n}_{2^{nd}} \cdots \underbrace{\alpha_1 \alpha_2 \dots \alpha_n}_{\mu_i{}^{th}} \rangle^*$. Decision point constructs are added based on the given PSOS s_i. The folding optimization groups the identified decision states of the normalized PSOS s'_i in the following manner: the actor α_j from PSOS s_i is marked if a decision state is identified at least in one of the corresponding states of the μ_i firings of the actor α_j in the normalized PSOS s'_i. The corresponding state related to firing of the marked actor α_j in PSOS s_i is considered as decision state.

The corresponding state related to firing of the actor α_j in PSOS s_i which is considered as a decision state imposes a decision state to all μ_i corresponding states of actor α_j in PSOS s'_i. So, no decision state will be lost after decision state folding and the only effect is introducing unnecessary decision state controlling. We need to show that this extra controlling does not effect the execution of the SDFG according to the schedule. The construct added in a decision state is used to guarantee execution of the actor of choice of that decision state. This does not violate the actor firing order according to the PSOS in that state. It only forces the only enabled actor in that state to be fired. □

DSM adds some components per decision state to enforce the firing of the enabled actor in a decision state which is in line with the given PSOS (i.e., actor of choice). Proposition 6 explains how those components can guarantee the firing of the actor of choice in the decision state.

Proposition 6. *The PSOS s_i is a schedule for actors $A_i \subseteq A$ from SDFG (A,C). Let $\omega_j \in \Omega$ be a decision state within PSOS s_i and $\Delta_j \subseteq A_i$ the set of the opponent actors in decision state ω_j. The actor of choice in decision state ω_j (denoted by actor a_c) is the only actor which can fire in decision state ω_j among all actors in Δ_j after applying DSM and the periodic behavior of the SDFG is also preserved.*

Proof. We need to show that the opponent actors $\Delta_j \setminus \{a_c\}$ can not be enabled in the decision state ω_j after applying DSM. Accordingly, the actor of choice a_c in the decision state ω_j is the only actor which can fire in decision state ω_j among all actors in Δ_j.

In the DSM technique, actor $a_{i-\omega_j}$ is added for each decision state $\omega_j \in \Omega$ within the PSOS s_i. An opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ in the decision state ω_j is dependent on the new actor $a_{i-\omega_j}$ because of the added channel $c_{i-a_k\omega_j}$; this channel is initialized with $BEF(a_k, \omega_j, s_i)$ tokens. The new actor $a_{i-\omega_j}$ is also dependent on the actor of choice a_c of the decision state ω_j because of the added channel $c_{i-a_c\omega_j}$; this channel is initialized with $AFT(a_c, \omega_j, s_i)$ tokens. For each opponent actor $a_k \in \Delta_j$, a channel is added between the actor $a_{i-\omega_j}$ and the opponent actor a_k. The rate of the added channel on the side of the actor a_k (actor $a_{i-\omega_j}$) is equal to one ($CNT(a_k, s_i)$).

An opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ fires $BEF(a_k, \omega_j, s_i)$ times before decision state ω_j and every time the opponent actor a_k consumes one token from channel $c_{i-a_k\omega_j}$. So, the $BEF(a_k, \omega_j, s_i)$ firings of actor a_k before state ω_j consume all tokens which were available in channel $c_{i-a_k\omega_j}$. Hence, the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ cannot fire in state ω_j. Firings of the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ from decision state ω_j onward will be dependent on the firing of the actor $a_{i-\omega_j}$ to provide the required tokens in channel $c_{i-a_k\omega_j}$. As mentioned before, the actor $a_{i-\omega_j}$ depends on the actor of

choice a_c. So, firings of the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ from decision state ω_j onward cannot happen before firing of the actor of choice a_c of the decision state ω_j in that state. Hence, the actor of choice a_c is the only actor among the other opponent actors in state ω_j which can fire. The actor of choice a_c is fired $BEF(a_c, \omega_j, s_i)$ times by state ω_j and this results in $BEF(a_c, \omega_j, s_i)$ tokens being produced in channel $c_{i-a_c\omega_j}$; as channel $c_{i-a_c\omega_j}$ is initialized with $AFT(a_c, \omega_j, s_i)$ tokens, the number of tokens in this channel is $BEF(a_c, \omega_j, s_i) + AFT(a_c, \omega_j, s_i) = CNT(a_c, s_i)$ after firing actor a_c in decision state ω_j. So, there will be sufficient tokens (for one firing of the actor $a_{i-\omega_j}$) on the only channel leading to actor $a_{i-\omega_j}$ after firing of the actor of choice a_c in decision state ω_j. Then, firing of actor $a_{i-\omega_j}$ consumes all $CNT(a_c, s_i)$ tokens that are present in channel $c_{i-a_c\omega_j}$ and it produces $CNT(a_k, s_i)$ tokens in channel $c_{i-a_k\omega_j}$ ($a_k \in \Delta_j \setminus \{a_c\}$); therefore, the opponent actors $\Delta_j \setminus \{a_c\}$ are not any more dependent on the actor $a_{i-\omega_j}$ in the remainder of the current iteration of the PSOS s_i.

The firings of the opponent actor $a_k \in \Delta_j \setminus \{a_c\}$ after decision state ω_j consumes $AFT(a_k, \omega_j, s_i)$ tokens from channel $c_{i-a_k\omega_j}$; as a result, at the end of the PSOS iteration, the token amount on this channel returns to its initial value which is $BEF(a_k, \omega_j, s_i)$ (because $BEF(a_k, \omega_j, s_i) = CNT(a_k, s_i) - AFT(a_k, \omega_j, s_i)$). The actor of choice a_c fires $AFT(a_c, \omega_j, s_i)$ times after decision state ω_j and the number of tokens in channel $c_{i-a_c\omega_j}$ returns to $AFT(a_c, \omega_j, s_i)$. These initial token resettings at the end of the PSOS iteration ensure the periodic behavior for the added components in each decision state. Thus, in a decision state only the actor of choice (which is in line with the given schedule) amongst all opponent actors of the decision state can fire and this eliminates any uncertainty because of the decision state. □

The following theorems state the correctness of DSM in modeling a single PSOS for a subset of the actors of the SDFG.

Theorem 1. *Consider PSOS s_i as a schedule for actors $A_i \subseteq A$ from SDFG G (A, C). For any execution σ' of $G'(A', C')$ it holds that σ satisfies s_i where it is assumed that σ is the execution of $G(A, C)$ with $\texttt{orderList}(\sigma, A) = \texttt{orderList}(\sigma', A)$.*

Proof. Proposition 6 states that in a decision state of PSOS s_i, an enabled actor of the decision state which is in line with PSOS s_i is the only actor able to fire in that state among all enabled actors in A_i. So, the order of s_i is the only possible order of actor firing for those actors of the SDFG G' in the set A_i. Proposition 2 implies that the next PSOS iteration cannot interfere. Hence, for any execution σ' of SDFG $G'(A', C')$, $\texttt{orderList}(\sigma', A_i)$ has the form of $(s_i)^\kappa$ where $\kappa \in \mathbb{N}$ (i.e., infinite repetition of s_i). It is assumed that $\texttt{orderList}(\sigma, A) = \texttt{orderList}(\sigma', A)$; as $A_i \subseteq A$, we can conclude that $\texttt{orderList}(\sigma, A_i) = \texttt{orderList}(\sigma', A_i)$. Hence, $\texttt{orderList}(\sigma, A_i)$ also has the form of $(s_i)^\kappa$ and this form satisfies s_i; in other words, σ satisfies s_i. □

Theorem 2. *Consider PSOS s_i as a schedule for actors $A_i \subseteq A$ from SDFG $G(A, C)$. For any execution σ of $G(A, C)$ that satisfies s_i it holds that there is exactly one σ' that is an execution of $G'(A', C')$ such that $\texttt{orderList}(\sigma, A) = \texttt{orderList}(\sigma', A)$.*

Proof. DSM adds actors A_{s_i} and channels C_{s_i} to model the PSOS s_i in the SDFG $G(A, C)$.

It is assumed that the firing order of actors belonging to the set A in execution σ' has the same actor firing order as in execution σ and execution σ satisfies the PSOS s_i. We need to show that there is precisely one execution with the property of execution σ' and that is a valid execution for SDFG G'. In a precise way, the actor firing order related to the set A in execution σ (i.e., orderList$(\sigma, A) = \{\alpha_1, \alpha_2, \cdots\}$) is a possible actor firing order for the original actors (i.e., actors not added by DSM) of the SDFG G' when σ' is an execution of the SDFG G'. Actor α_x from orderList(σ, A) belongs either to A_i or to $A_o = A \setminus A_i$ ($x \in \mathbb{N}$). The state transition $\omega_x \xrightarrow{\alpha_x} \omega_{x+1}$ in execution σ is related to the firing of actor α_x. The state transition $\omega'_y \xrightarrow{\alpha_y} \omega'_{y+1}$ in execution σ' is related to the firing of actor α_y. The difference between states from execution σ and σ' is only in the extra channels added by DSM (i.e., C_{s_i}). Any channel from C_{s_i} is connected to an actor from A_i; in other words, it is not connected to any actor from A_o. Incoming channels of an actor determine whether that actor can fire or not. Consider state ω'_y from execution σ' has the same content of state ω_x from execution σ for all channels in C. So, when an actor α_x belongs to A_o, it means that actor α_x which can fire in state ω_x of execution σ can fire in state ω'_y of execution σ' (i.e., α_y is α_x) because the content of the state related to channels C_{s_i} has no influence on actor enabling in that particular state for any actor from A_o. But, when an actor α_x belongs to A_i, the content of the state related to channels C' could manipulate the actor firing order. As we assume firings of actor α_x in execution σ satisfy the PSOS s_i, actor α_x can fire in the corresponding state (i.e., ω'_y) of execution σ' (i.e., α_y is α_x) because components added by DSM force the firing of the actor which is in line with the given PSOS s_i among all actors in A_i (see Proposition 6) and it is assumed that α_x is in line with PSOS s_i. So, the firing order of actors from $A' \setminus A_{s_i}$ in execution σ' follows the same firing order as it is indicated in execution σ. Each actor $a \in A_{s_i}$ also has a single possible firing order in each PSOS iteration; if a is added to control the actor firing in a decision state, it fires before the actor of choice in the decision state (see Proposition 6) and if it is added for the sake of inter-iteration prevention purpose, it fires at the end of the PSOS iteration (see Proposition 2). Hence, there exists only one possible firing order in execution σ' for each actor $a \in A_{s_i}$. So, all actors from A' have exactly one firing order in execution σ' where $\texttt{orderList}(\sigma, A) = \texttt{orderList}(\sigma', A)$. □

The size of the schedule-extended graph (e.g., SDFG G') is dependent on the number of decision states found in the given schedule (e.g., PSOS s_i). In this section, decision state

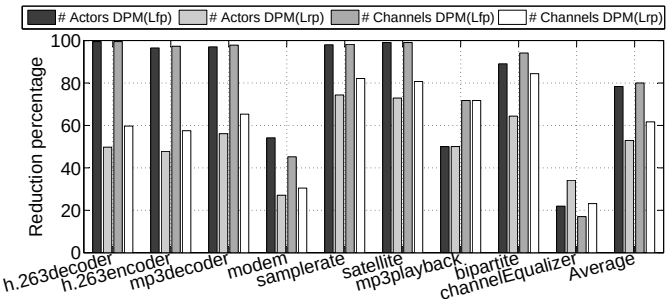
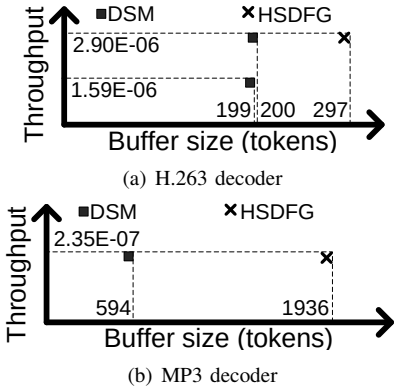


Fig. 7. Reduction in the size of the schedule-extended graphs when using DSM in contrast to the HSDFG-based technique (Higher is better). Schedules are generated by list forward priorities (Lfp) and list reverse priorities (Lrp).



(a) H.263 decoder

(b) MP3 decoder

Fig. 8. Pareto space of schedule-extended graphs modeled by DSM and HSDFG-based techniques (the scales of the two graphs are different).

identification for a sub-set of actors of the SDFG G (i.e., $A_i \subseteq A$) which belong to the schedule of interest (i.e., s_i) is explained regardless of existing other schedules for the rest of the actors in the SDFG (i.e., actors in $A_o = A \setminus A_i$). In our implementation, we consider other possible schedules designated for the rest of the actors (i.e., actors in A_o) to reduce the number of the decision states and as a result the size of the schedule-extended graph. As we explained in Sec. V-C2, actors which do not belong to schedule s_i should fire according to their schedule (if there is any) to perform their maximal execution. Any actor $a_o \in A_o$, which belongs to another PSOS s_j ($j \neq i$), is fired in function $maxExec$ of the DSM algorithm when (1) it is enabled and (2) its firing satisfies s_j. Without the second condition, firing of a_o could enable an actor from PSOS s_i and lead to unnecessary decision states. So, considering other schedules in the function $maxExec$ of DSM algorithm removes such redundant decision states.

VII. Experimental results

We used a set of DSP and multimedia applications to assess our DSM technique. The following SDFGs are extracted from realistic applications: modem [1], sample-rate converter [1], satellite receiver [18], mp3playback [19], channel equalizer [20], H.263 decoder [10], H.263 encoder [21], and MP3 decoder [10]. We also consider the bipartite SDFG [18] which is a commonly used artificial SDFG.

A PSOS determines the actor firing order and as such it influences the enabled actors in a state; as a result, the number of decision states can be different for different PSOSs. The size of the schedule-extended graph using DSM depends on the number of decision states in the given schedules. So, the compactness of the schedule-extended graph depends on the input schedule which should be modeled. We use the common list scheduler [22] to determine the PSOSs for the applications. We use two different variations of list scheduling to verify DSM in different situations. The first list schedule uses forward priorities (Lfp) and the second one uses reverse priorities (Lrp). Actors closer to the inputs of the graph have higher priority in the Lfp schedules compared to actors closer to the outputs of the graph and vice-versa in Lrp schedules.

Fig. 7 shows the reduction percentage in the size of the schedule-extended graph when using DSM in contrast to the HSDFG-based technique. Using schedules generated by Lfp, the number of decision states is less than when Lrp is used, except in the channel equalizer and mp3playback applications. By using Lfp scheduling, actors closer to inputs have higher priority compared to actors closer to outputs. This leads to consecutive execution of an actor followed by consecutive execution of another actor with lower priority and so on. Thanks to our optimization in DSM, considering only one decision state before a context switch will be sufficient (e.g., decision state ω_9 in Fig. 4) and the number of decision states can be reduced significantly. Usually actors closer to outputs are dependent on actors closer to inputs in an SDFG; this dependency can prevent an actor from being executed consecutively in a graph scheduled by Lrp. As a result of that, the number of context switches in a graph scheduled by Lrp will typically be larger compared to Lfp. Hence, the effectiveness of the decision state optimization in DSM reduces and extra elements are required to model the schedules in the graph. The exceptions in the channel equalizer and mp3playback are due to the existence of a cycle in the SDFG; the cycle can increase the number of context switches in the schedule and as a result, Lfp could result in the same or a higher amount of decision states in DSM compared to Lrp. Important, however, is that in our experiments, DSM always outperforms the HSDFG-based technique regardless of the input schedule. The number of actors (channels) using DSM is 66% (71%) lower compared to the HSDFG-based technique on average, 99% (99%) lower in the best-case and 28% (20%) lower in the worst-case observed in our experiments. Besides the compactness of the schedule-extended graph, DSM preserves the original structure of an SDFG which is not guaranteed for the state of the art technique.

To further analyze the effectiveness of DSM, we applied a buffer sizing algorithm from [10] on the schedule-extended SDFGs of the H.263 decoder and MP3 decoder applications. The H.263 decoder is mapped on a platform with two processors. The actor *vld* and *iq* are mapped on the first processor with a PSOS $\langle vld(iq)^{99}\rangle^*$ and the actor *idct* and *mc* are mapped on the second processor with a PSOS $\langle (idct)^{99} mc\rangle^*$. The analysis time for buffer sizing on the schedule-extended

H.263 decoder is less than 1 ms when using DSM to model the schedules. The same analysis lasts for 1330 ms when using the technique from [11] to model the same schedules in the same graph. Fig. 8(a) shows the complete design space (Pareto space) of throughput and buffer size when modeling the schedule with DSM and the HSDFG-based technique [11]. A single channel in an SDFG corresponds to a set of channels in the equivalent HSDFG. As a result, the buffer sizing technique cannot find the minimal buffer size when applying it on the equivalent HSDFG. Our experiments show these inaccuracies. Applying buffer sizing on the graph which models the schedules using the technique from [11] results in 43% overestimation in required buffer space compared to applying the same buffer sizing technique on the graph which models the same schedules when using our technique. Fig. 8(b) shows results for the MP3 decoder. We used the mapping and scheduling from [12] which maps the MP3 decoder on a platform with 3 processors. The analysis time on the graph which models the schedule using our technique is 594 ms while 141610 ms is required to perform the same analysis on the graph using the technique from [11]. Using the technique from [11] results in 2.26 times overestimation in buffer size compared to using our technique.

Modeling a PSOS in an SDFG using DSM requires executing one complete SDFG iteration. The number of states in one iteration could be exponential in the number of actors in the graph. However, for all real-world SDFGs used in our experiments, the execution time of the DSM is below 1 ms.

VIII. Conclusion

We presented a technique, DSM, to model periodic static-order schedules directly in an SDFG. The resulting graphs are much smaller (often much less than half the size) than graphs resulting from the state of the art technique that first converts an SDFG to an HSDFG. This results in a speed-up of performance analysis. Computing the trade-off between buffering and throughput for multi-processor implementations, for example, becomes several orders of magnitude faster. Moreover properties like buffer sizes can be analyzed more accurately. For future work, we would like to investigate to further optimize the models for some specific scheduling classes, e.g., single appearance schedules.

References

[1] S. S. Bhattacharyya *et al.*, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing*, vol. 21, pp. 151–166, 1999.

[2] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[3] P. Poplavko *et al.*, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," CASES. ACM, 2003, pp. 63–72.

[4] M.-Y. Ko *et al.*, "Compact procedural implementation in DSP software synthesis through recursive graph decomposition," SCOPES. ACM, 2004, pp. 47–61.

[5] A. Bonfietti *et al.*, "Throughput constraint for synchronous data flow graphs," CPAIOR. Springer-Verlag, 2009, pp. 26–40.

[6] S. Stuijk *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," DAC. ACM, 2007.

[7] W. Liu *et al.*, "Efficient SAT-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization," RTSS. IEEE, 2008, pp. 492–504.

[8] Y. Yang *et al.*, "Automated bottleneck-driven design-space exploration of media processing systems," DATE. ACM, 2010, pp. 1041–1046.

[9] A. Ghamarian *et al.*, "Throughput analysis of synchronous data flow graphs," ACSD. IEEE, 2006, pp. 25–36.

[10] S. Stuijk *et al.*, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.

[11] N. Bambha *et al.*, "Intermediate representations for design automation of multiprocessor DSP systems," *Design Automation for Embedded Systems*, vol. 7, no. 4, pp. 307–323, 2002.

[12] M. Geilen and S. Stuijk, "Worst-case performance analysis of synchronous dataflow scenarios," CODES+ISSS. ACM, 2010, pp. 125–134.

[13] M. H. Wiggers *et al.*, "Monotonicity and run-time scheduling," EMSOFT. ACM, 2009, pp. 177–186.

[14] H. H. Wu *et al.*, "A model-based schedule representation for heterogeneous mapping of dataflow graphs," HCW. IEEE, 2011, pp. 66–77.

[15] S. Bhattacharyya *et al.*, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[16] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceeding of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[17] M. Geilen *et al.*, "Minimising buffer requirements of synchronous dataflow graphs with model checking," DAC '05. ACM, 2005, pp. 819–824.

[18] S. Ritz *et al.*, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," ICASSP. IEEE, 1995.

[19] M. H. Wiggers *et al.*, "Efficient computation of buffer capacities for cyclo-static dataflow graphs," DAC. ACM, 2007, pp. 658–663.

[20] A. Moonen *et al.*, "Practical and accurate throughput analysis with the cyclo static dataflow model," MASCOTS. IEEE, 2007, pp. 238–245.

[21] H. Oh and S. Ha, "Fractional rate dataflow model for efficient code synthesis," *Journal of VLSI Signal Processing*, vol. 37, pp. 41–51, 2004.

[22] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill, 1994.