Schedule-Extended Synchronous Dataflow Graphs

Morteza Damavandpeyma, *Student Member, IEEE,* Sander Stuijk, Twan Basten, *Senior Member, IEEE,* Marc Geilen, *Member, IEEE,* and Henk Corporaal, *Member, IEEE*

***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 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 techniques to model two types of static-order schedules, i.e., periodic schedules and periodic single appearance schedules, directly in an SDFG. Experiments show that both techniques produce 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.**

***Index Terms*—Synchronous dataflow graphs, periodic schedules, single appearance schedules, schedule modeling.**

I. Introduction

Synchronous dataflow graphs (SDFGs) are widely used to model digital signal processing (DSP) and multimedia applications [1]–[4]. Model-based design-flows (e.g., [1], [5]–[8]) model binding and scheduling decisions into the SDFG. This enables the analysis of performance properties (e.g, throughput [9]) or resource requirements (e.g., buffer sizes [10]) under resource constraints. Figure 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 a fixed number of tokens from its input edges and produces a fixed number of tokens on its output edges. These numbers 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. An SDFG can be bound to a multi-processor platform. Each processor in the platform executes a set of actors from the SDFG; the firings of actors bound to a processor are required to be sequentialized. For this purpose, a finite periodic schedule can be constructed. Such a schedule is called a *periodic static-order schedule* (*PSOS*). PSOSs only specify the firing order of actors. This separates them from fully static schedules, which determine absolute start times of actors (e.g., schedules generated using the technique of [11]). Traditionally, for DSP software synthesis, a sub-set of all periodic static-order schedules is considered. This sub-set contains so-called *single appearance schedules* (SAS) [1]. In a SAS, the functional code of the actors is included in a nested loop structure such that each piece of code occurs only once. This minimizes the code size potentially at the cost of additional buffer memory needed to implement the channels. A model-based design-flow usually uses PSOSs (or a sub-set of PSOSs such as SASs) for an application modeled with an SDFG. In this way timing (throughput) and memory usage (buffers) can be analyzed.

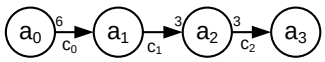


Fig. 1. An example SDFG.

There is only one technique [12] 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 one [2]. Figure 2 (without the colored edges) shows the equivalent HSDFG of the SDFG in Figure 1. The technique of [12] sequentializes the actor firings by inserting a channel between each pair of consecutive actors in a schedule. At the end of a schedule, it adds a channel with one initial token from the last to the first actor in the schedule. This ensures an indefinite execution of the graph according to the schedule. 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 technique of [12] adds in total 15 channels to the HSDFG of the example graph (the green edges for s_0 and the blue edges for s_1 in Figure 2). For example, s_0 indicates an indefinite sequence of one firing of a_0 followed by two firings of a_2. This order is enforced in the HSDFG of Figure 2 by the green edges between the actors a_{0_1}, a_{2_1}, and a_{2_2}.

The SDFG to HSDFG conversion can lead to an exponential increase in the size of the graph. For example, such a conversion for an H.263 decoder [10] (with QCIF resolution) increases the graph size from 4 actors to 1190 actors. Note that the number of actors in the resulting HSDFG highly depends on how the application is modeled in the original SDFG. The run-time of SDFG analysis algorithms depends amongst others on the size of the graph. As a result, the run-time of many

Manuscript received Nov. 6, 2012; revised Feb. 14, 2013.

The authors are with the Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands (e-mail:{m.damavandpeyma, s.stuijk, a.a.basten, m.c.w.geilen, h.corporaal}@tue.nl).

T. Basten is also with TNO, Embedded Systems Innovation, Eindhoven, The Netherlands.

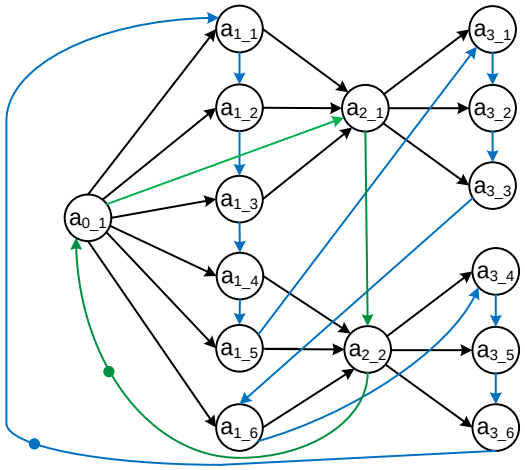


Fig. 2. PSOSs $s_0 = \langle a_0 (a_2)^2 \rangle^*$ and $s_1 = \langle (a_1)^5 (a_3)^3 a_1 (a_3)^3 \rangle^*$ modeled in the SDFG of Figure 1 using the technique from [12]; each a_{i_j} actor in the HSDFG is an instance of SDFG actor a_i.

SDFG analysis algorithms may increase drastically when modeling PSOSs in the graph using the technique from [12]. 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 [12], the same analysis lasts 1330 ms. SDFG analysis algorithms are usually repeated more than once in an iterative design-flow. As an example, for the SDFG of an H.263 decoder, the design-flow from [6] performs 8 throughput calculations on the SDFG to obtain the desired binding. Hence, it is vital to maintain a compact *schedule-extended graph*, i.e., a graph in which schedules are modeled explicitly, to provide a fast and practical design flow. There is a second drawback to the technique from [12]. The original graph structure is lost due to the conversion to an HSDFG. A single channel in an SDFG corresponds to a set of channels in the HSDFG. In Figure 2, for example, the six edges between actor a_{0_1} and the a_{1_j} actors correspond to the single edge between a_0 and a_1 in the SDFG of Figure 1. 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 49% when applying the technique of [10] to the HSDFG.

A novel technique is needed to model PSOSs 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. In [13], we presented a schedule modeling technique, called DSM, to model any PSOS directly in an SDFG. In this paper, DSM is discussed in more detail. In addition, a second schedule modeling technique, called SASM, is introduced that is limited to SASs, but that results in more compact models compared to the first technique when modeling SASs. Correctness of the two approaches is formalized. Extensive experiments are carried out for evaluation purposes.

DSM and SASM can be used in any model-based design-flow that models PSOSs into the SDFG (e.g., [1], [5]–[8]). Conversion to an HSDFG may be inevitable at some steps of a design trajectory. For example, multi-processor scheduling may require such conversions, although some techniques exist that can find schedules for SDFGs without any conversion to HSDFGs. For example, the technique presented in [14] solves the buffer sizing and scheduling problems simultaneously at the SDFG level. It is not the conversion from SDFG to HSDFG itself that is problematic though. The problem is that analyses or optimizations on large HSDFGs may be time consuming (e.g. throughput analysis) or inaccurate (e.g. buffer sizing). With our techniques, obtained schedules can be annotated back to the original SDFG; hence, the later analysis and optimization can be performed on the schedule-extended SDFG. Besides the already mentioned analyses, also for example dynamic voltage scaling can be directly applied to a schedule-extended SDFG model of an application mapped to a multi-processor platform [15]. Note that code generation is another step which requires an SDFG to HSDFG conversion; this conversion can be delayed until all (or most of the) prior analyses are carried out on the SDFG. Ultimately, the proposed techniques may save significant amounts of analysis time in a multi-processor design flow and they may lead to more accurate results.

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

II. Related Work

The technique from [12] is the only available technique to model PSOSs in an SDFG, through a conversion to an HSDFG. As already explained, this technique may result in a long run-time for analysis algorithms and/or inaccurate results from these algorithms. Our techniques alleviate both shortcomings of the technique from [12].

The work in [16] models the effect of a budget scheduler or preemptive TDMA scheduler on the temporal behavior of the SDFG, either by computing an accurate worst-case response time or, more precisely, by introducing additional actors to model the timing impact as 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 force 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 [16]. However, for non-preemptive schedules, our results are tighter and our techniques require less analysis time. The authors of [17] have shown that an SDFG can be used to consider an application with resource sharing possibilities; they perform buffer sizing under a throughput constraint considering a given schedule for the actors using a shared resource. For shared resource analysis, they use event-models [18] which is based

on realtime-calculus [19]. Our approach differs from [17], since modeling schedules directly into SDFGs enables us to use dedicated analysis for dataflow graphs. Moreover, the technique of [17] can only handle an SDFG with limited types of cycles, such as cycles formed by self-edges or the back edges modeling the buffer capacity between two actors. However, staying in dataflow domain, as is done in our technique, does not impose such a limitation on the graph structure.

Ref [20] uses some new (custom) components, e.g., $if-then-else$, to model schedules in an SDFG. These components are not supported by the common model-based design-flows using SDFGs (e.g., [1], [5]–[8]) and cannot be modeled in an SDFG using the basic elements of an SDFG (i.e., actors and channels). Our techniques eliminate the need for any new (custom) component. As a result, any analysis technique for SDFGs is directly applicable to the schedule-extended SDFG.

III. Synchronous Dataflow Graphs

Let $\mathbb{N}$ denote the positive natural numbers and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Consider $Ports$ as a set that contains all ports; each port $p \in Ports$ has a finite rate $Rate(p) \in \mathbb{N}$. An actor a_i is a tuple (In, Out) consisting of a set $In \subseteq Ports$ of input ports and a set $Out \subseteq Ports$ of output ports with $In \bigcap Out = \emptyset$.

Definition 1. (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 an actor and the channel destination is an input port of an actor. Each port of an actor is connected to only one channel and each channel end is connected to a single port. For every actor $a_i = (In, Out) \in A$, $InC(a_i)$ represents all channels connected to the ports in In and $OutC(a_i)$ represents all channels connected to the ports in Out.*

The SDFG of Figure 1 has four *actors* ($A = \{a_0, a_1, a_2, a_3\}$) and three *channels* ($C = \{c_0, c_1, c_2\}$). Actors communicate with *tokens* sent from one actor to another over the channels. Tokens are 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 a fixed number of tokens from its input edges and produces a fixed number of tokens on its output edges. These numbers are called the *rates* (indicated next to the channel ends when the rates are larger than 1). 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* [1] of an SDFG. The repetition vector determines the number of times each actor should be fired in order to bring the SDFG back to its initial token distribution. Notation $\gamma(a)$ refers to the repetition vector value of actor a. The repetition vector of the SDFG shown in Figure 1 is $\gamma = [1\ 6\ 2\ 6]^T$. It corresponds to 1 firing of a_0, 6 firings of a_1, 2 firings of a_2 and 6 firings of a_3. Channels can contain different numbers of tokens. A *state* of an SDFG (represented by ω) is described by the number of tokens in all channels of the SDFG. We assume that the initial state of an SDFG is given by an initial token distribution ω_0. An actor $a_i \in A$ is *enabled* in an SDFG state ω_j iff $\omega_j(c) \geq Rate(q)$ for each channel $c = (p, q) \in InC(a_i)$. When an actor a_i starts its firing, it removes $Rate(q)$ tokens from all $(p, q) \in InC(a_i)$ and when it ends, it produces $Rate(p)$ tokens on every $(p, q) \in OutC(a_i)$. Consistency (i.e., the existence of a repetition vector) and absence of deadlock are necessary in practice for SDFGs and can be verified efficiently [21], [22]. Any SDFG which is not consistent requires unbounded memory to execute or deadlocks. Therefore, we limit ourselves to consistent and deadlock-free SDFGs.

IV. SDFG Static-Order Scheduling

Assume that the initial state ω_0 for the SDFG of Figure 1 is equal to $(c_0, c_1, c_2) \rightarrow (0, 0, 0)$. Actor a_0 is enabled in ω_0. Firing a_0 results 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 2. (Execution) *An execution σ of an SDFG is an infinite alternating sequence of states and transitions $\omega_0 \xrightarrow{a_i} \omega_1 \xrightarrow{a_j} \cdots$ starting from a 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 order the firings of the enabled actors on the processor. The fixed port rates make it possible to statically schedule SDFGs with a finite schedule per processor. Such a schedule orders the actor firings on the underlying processor. This type of schedules, which are called *periodic static-order schedules (PSOSs)*, can be repeated indefinitely. A separate PSOS should be constructed for each processor. Each PSOS only includes actors bound to this specific processor. The following definition is used to formally specify a PSOS.

Definition 3. (Periodic Static-Order Schedule (PSOS)) *A PSOS is a finite ordered list of (a sub-set of) actors in an SDFG (A, C). A PSOS is denoted by $s_i = \langle \alpha_1 \alpha_2 \ldots \alpha_n \rangle^*$ where each $\alpha_j|_{1 \leq j \leq n}$ is a sub-schedule that 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 ($A_i \subseteq A$).*

A PSOS can be represented in a compact format, called a looped schedule (LS).

Definition 4. (Looped Schedule (LS)) *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 α_j is either an actor firing or a (nested) looped schedule and $\beta_j \in \mathbb{N}$.*

Definition 5. (Single Appearance Schedule (SAS)) *A LS in which each actor appears only once is called a single appearance schedule (SAS).*

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

PSOS s_0 is a SAS while PSOS s_1 is not. PSOS $s_1' = \langle (a_1)^3 (a_3)^3 \rangle^*$ can be used as a SAS for actors a_1 and a_3.

Definition 6. (SDFG ITERATION) *Assume SDFG (A,C) has repetition vector γ. An SDFG iteration is a set of actor firings such that for each $a \in A$, the set contains $\gamma(a)$ firings of a.*

Definition 7. (PSOS ITERATION) *Let $s_i = \langle \alpha_1 \alpha_2 \dots \alpha_n \rangle^*$ be a PSOS that schedules actors in $A_i \subseteq A$. A PSOS iteration is a sequence of actor firings following the actor order specified in s_i starting from actor α_1 and ending with actor α_n with a length equal to $|s_i|$ and including only actors from A_i.*

The actor firing order in an execution $\sigma = \omega_0 \xrightarrow{a_x} \omega_1 \xrightarrow{a_y} \cdots$ can be captured using a list $\langle a_x, a_y, \cdots \rangle$ where the j^{th} element in this list is the actor which is fired in the transition from ω_{j-1} to ω_j. The notation $orderList(\sigma, A_i)$ represents the mentioned list where actors which do not belong to A_i are omitted. For example, in the SDFG of Figure 1, consider an execution σ that results in list $\langle a_0, a_1, a_1, a_1, a_2, a_1, a_1, a_3, a_3, a_3, a_1, a_2, a_3, a_3, a_3 \rangle$ with $A_1 = \{a_1, a_3\}$; then $orderList(\sigma, A_1) = \langle a_1, a_1, a_1, a_1, a_1, a_3, a_3, a_3, a_1, a_3, a_3, a_3 \rangle$. We say that the corresponding execution of an SDFG satisfies a PSOS when the SDFG is executed according to the PSOS. We use the following to formalize this term.

Definition 8. (SATISFACTION) *Let σ be an execution of an SDFG (A, C) and s_i a PSOS for actors $A_i \subseteq A$. If it exists, let σ' be the prefix of σ such that it contains exactly $\gamma(a_i)$ occurrences of actor $a_i \in A_i$; σ' covers σ precisely up to the point that one PSOS iteration is executed. Execution σ satisfies PSOS s_i iff σ' exists and the ordered list $orderList(\sigma', A_i)$ corresponds to the order specified in s_i.*

When an execution of a consistent and deadlock-free SDFG satisfies the specified PSOSs, the channels of the SDFG need bounded memories (according to Theorem 1 from [23]). 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 9. (NORMALIZED PSOS) *A PSOS s_i is called normalized iff 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., $r = \frac{u}{v}$ with $u = 1$ and $v \in \mathbb{N}$), although our technique can also be directly applied to model other PSOSs (i.e., in which $u \in \mathbb{N}$).

V. MODELING PERIODIC STATIC-ORDER SCHEDULES

In this section, we introduce a technique to model PSOSs in an SDFG. We first illustrate all ingredients of our approach through a running example, and then discuss the algorithm and the main steps in the algorithm in more detail. Note that a schedule is correctly modeled if and only if any execution of the schedule-extended graph satisfies the schedule and if any execution of the original graph that satisfies the schedule is still feasible in the schedule-extended graph.

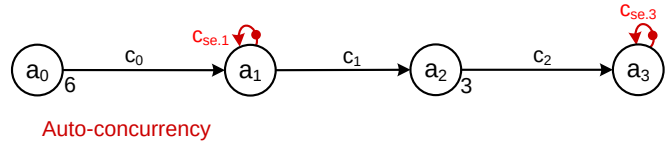
A. Running example

Here we briefly introduce our approach in modeling a PSOS in an SDFG using a running example. For this purpose consider the example SDFG shown in Figure 1 and the PSOS $s_1 = \langle (a_1)^5 (a_3)^3 a_1 (a_3)^3 \rangle^*$ which is a schedule for a_1 and a_3. Our approach captures this schedule in the SDFG in three steps, that (i) remove auto-concurrency, (ii) avoid inter-iteration execution, and (iii) enforce the correct scheduling decisions.

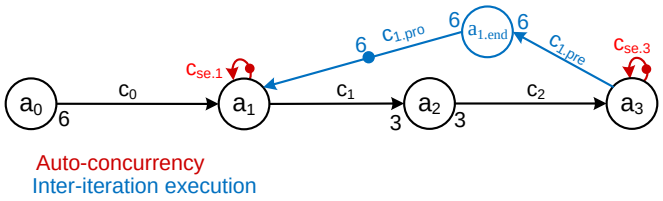
In the example SDFG, a single firing of a_0 produces 6 tokens in channel c_0; then 6 firings of a_1 can be performed simultaneously. This simultaneous firing of an actor is called *auto-concurrency*; in practice, this corresponds to executing multiple instances of a function (task) in parallel. Auto-concurrency for an actor cannot be handled in a real hardware platform, unless more than one processor is allocated for that actor. In this work, we focus on the case that an actor is mapped to one processor. Hence, auto-concurrency must be removed from the model. To sequentialize firings of an actor, a self-edge with one initial token must be added to that actor; this way auto-concurrency can be removed for that actor. Figure 3(a) shows the example SDFG of Figure 1 in which any auto-concurrency related to a_1 and a_3 is removed by adding two self-edges (shown in red).

In one PSOS iteration, each actor must fire a specific number of times. Actors must not be able to get fired more than the number of times indicated by the PSOS. Consider the following situation in the example SDFG of Figure 1. Two firings of a_0 provide 12 tokens in channel c_0; this number of tokens is enough for 12 firings of the actor a_1. The first 6 firings of a_1 belong to the first iteration and the second 6 firings belong to the second iteration. The second 6 firings of a_1 can occur before the completion of the first PSOS iteration of s_1; in this case, the resulting execution does not satisfy the given PSOS s_1. This situation is called *inter-iteration execution*. To prevent inter-iteration execution related to s_1, we create a dependency from the last actor appearing in s_1 to the first actor appearing in s_1; see the blue elements in Figure 3(b). This dependency limits the firing of the first actor to a number of firings required in one PSOS iteration.

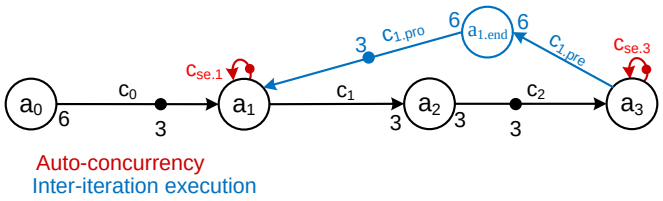
In the SDFG of Figure 3(b), consider the case that actors a_0, a_1 and a_2 have fired 1, 3 and 1 times, respectively. The initial token distribution is changed to the distribution shown in Figure 3(c). In this graph, both a_1 and a_3 from PSOS s_1 are enabled; but, only the firing of a_1 must be granted at this point to form an execution that satisfies the given PSOS s_1. We call such a state in which several actors are enabled a *decision state*. Later on, a precise definition is given for this concept. According to schedule s_1, actor a_3 must get enabled after 5 firings of a_1. For this purpose, a dependency is created from a_1 to a_3 (shown with green actor and channels in Figure 3(d));



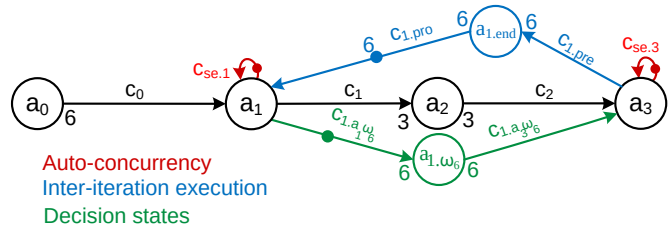
(a) Auto-concurrency has been removed.



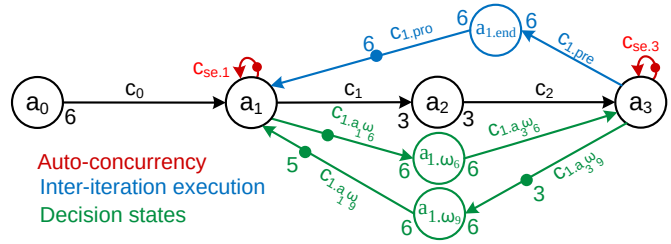
(b) Auto-concurrency and inter-iteration execution have been removed.



(c) SDFG of Figure 3(b) after 1, 3 and 1 firings of a_0, a_1 and a_2 resp., leading to a decision state in which both actors a_1 and a_3 are enabled.



(d) Creating a dependency for the first decision state.



(e) Creating a dependency for the second decision state.

Fig. 3. Step-by-step modeling of the PSOS s_1 in the SDFG of Figure 1.

this new dependency prevents a_3 from getting enabled unless a_1 has completed 5 firings. Another decision state can be found after a_1 has completed 5 firings; once again, both a_1 and a_3 from PSOS s_1 are enabled at this point; but, the firing of a_3 must take place. For this purpose, a dependency is created from a_3 to a_1 (see Figure 3(e)). This new dependency prevents a_1 from getting enabled from the current state onwards unless a_3 has completed 3 firings. The 5 initial tokens on the added input channel to a_1 ensure that the first 5 firings of a_1 can take place as planned. The SDFG of Figure 3(e) shows the final solution that models the PSOS s_1 in the SDFG of Figure 1.

B. The algorithm

Algorithm 1 captures our technique, called decision state modeling (DSM). DSM accepts an SDFG and one or several PSOSs as its input. In the algorithm $n+1$ ($n \in \mathbb{N}_0$) is the number of processors (or input PSOSs). DSM ensures that actor firings of each PSOS follow the specified order in that PSOS; the output of DSM is a new SDFG that models the provided PSOSs in the input SDFG. Figure 4 depicts the corresponding SDFG of Figure 1 which models the PSOSs s_0 and s_1 using DSM. The remainder of this section discusses the three main steps of the algorithm - removing auto-concurrency, avoiding inter-iteration execution, enforcing correct decisions in decision states - in detail.

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 source actor a_{src} to destination actor a_{dst}; the production rate of a_{src} on this channel is equal to $srcRate$ and the consumption rate of a_{dst} on this channel is equal to $dstRate$; this channel is initialized with $initTok$ initial tokens. The function $CNT(a_j, s_i)$ returns the number of times that the actor a_j is fired in one iteration of PSOS s_i. The function *BEF(a_k,j,s_i)* returns the number of times that a_k appears from the first position in the PSOS s_i to and including the j^{th} position in the PSOS s_i; the function *AFT(a_k,j,s_i)* returns the number of times that a_k appears from the $(j+1)^{th}$ position in the PSOS s_i to the last position in the PSOS s_i.

C. Auto-concurrency

An actor $a_i \in A$ can be enabled multiple times simultaneously in an SDFG state ω_j if $\omega_j(c) \geq k \cdot Rate(q)$ for each channel $c = (p, q) \in InC(a_i)$ where $k \in \mathbb{N}, k \geq 2$. This is called auto-concurrency. The firings of actor a_i should occur sequentially according to the PSOS to which 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 Figure 4, channels $c_{se.0} - c_{se.3}$ (shown in red) are used to prevent any auto-concurrency in the SDFG of Figure 1.

D. Inter-iteration execution

Consider actor a_0 in the SDFG of Figure 1; the 1^{st} firing of a_0 belongs to the 1^{st} PSOS iteration of s_0 and the 2^{nd} firing of a_0 belongs to the 2^{nd} PSOS iteration of s_0 and so on. Since a_0 has no input channel, it can always be fired regardless

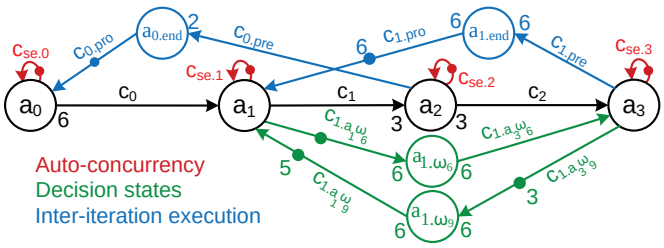


Fig. 4. PSOSs s_0 and s_1 modeled in the SDFG of Figure 1 using DSM.

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 Ω, $pos \leftarrow$ **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**$(a_k, pos[\omega_j], s_i)$)
17 **else**
18 **AC**(G, $c_{i.a_k\omega_j}$, $a_{i.\omega_j}$, a_k, **CNT**(a_k, s_i), 1, **BEF**$(a_k, pos[\omega_j], s_i)$)

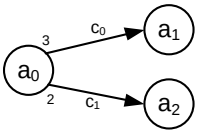


Fig. 5. An example SDFG.

of other actors in the graph. This behavior, which is called inter-iteration execution, can prevent an SDFG execution from satisfying the given PSOSs. Inter-iteration execution happens when one PSOS iteration has not been completed and an actor from that PSOS can proceed its firings beyond the current PSOS iteration. 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 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 Figure 4, 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 Figure 4.

E. Decision states

This sub-section presents the third step of DSM. It first defines the concept of a decision state and then proceeds with the algorithm for identifying decision states; after explaining two optimization steps, it ends with the technique to enforce the appropriate schedule decisions.

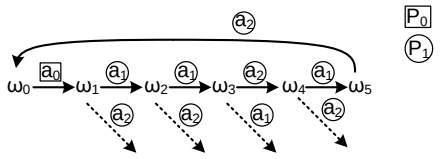


Fig. 6. The state space of the SDFG of Figure 5 when PSOSs $s''_0 = \langle a_0 \rangle^*$ and $s''_1 = \langle (a_1)^2 a_2 a_1 a_2 \rangle^*$ are used.

1) Concept: Multiple different actors mapped to a single processor may be enabled in a specific state. Here, we describe such situations in an SDFG execution. Consider the SDFG in Figure 5. Assume that a_0 is mapped to processor P_0 with PSOS $s''_0 = \langle a_0 \rangle^*$ and a_1 and a_2 are mapped to processor P_1 with PSOS $s''_1 = \langle (a_1)^2 a_2 a_1 a_2 \rangle^*$. For brevity, we only discuss the actors mapped to processor P_1. The corresponding state space - for one SDFG iteration - when executing our example SDFG using the PSOSs s''_0 and s''_1 is visualized in Figure 6. In this figure, the actors mapped to processor P_0 (P_1) are surrounded by a square (circle). Auto-concurrency and inter-iteration execution are excluded using the constructs introduced in Section V-C and Section V-D resp. The periodic behavior of the PSOSs is obvious from the state space where one SDFG iteration moves the graph to its initial state, i.e., ω_0 (see Figure 6). There are some states in Figure 6 in which more than one actor is enabled (e.g., $\omega_1 - \omega_5$) on processor P_1. 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 10. (DECISION STATE) *Consider the PSOS s_i as a schedule for 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 iff multiple different actors from A_i are enabled in ω_j.*

We use Ω to refer to the finite set containing all decision states occurring in the execution of an SDFG up-to one iteration of a PSOS. The following terminology describes the enabled actors in a decision state.

Definition 11. (OPPONENT ACTOR SET) *Let $\omega_j \in \Omega$ be a decision state for 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 fire. The following is used to describe such an actor.

Definition 12. (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 s_i. An actor $a_c \in \Delta_j$ is called the actor of choice of the decision state ω_j iff 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. Since a PSOS specifies a fixed firing order, there*

Algorithm 2: Get Decision States

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

```
1  ω_j ← the initial state of G
2  for i ← 1 to |s_c| do
3      ω_j ← maxExec(G, ω_j, {s_o1, ··· , s_on})
4      if sizeof(enabledActors(G, ω_j, s_c)) >1 then
5          pos[ω_j] ← i
6          Ω ← Ω ∪ {ω_j}
7          Δ_j ← enabledActors(G, ω_j, s_c)
8      ω_j ← fire(G, ω_j, s_c[i])
```

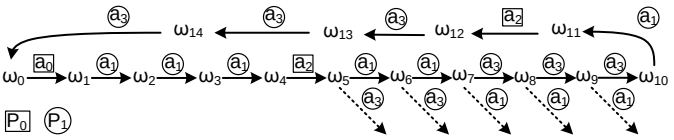


Fig. 7. The state space of the SDFG of Figure 1 when the PSOS $s_1 = \langle (a_1)^5 (a_3)^3 a_1 (a_3)^3 \rangle^*$ is the schedule of interest (i.e., s_c) in Algorithm 2.

can only be a single actor of choice in any decision state.

Lines 9-18 in DSM show how we deal with non-deterministic execution due to decision states. DSM models the given PSOSs one-by-one iteratively. The ordering of PSOSs in DSM does not have any impact on the final behavior. 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. At the same time, a value is preserved for each decision state that captures the relative position of that decision state with respect to the beginning of the schedule of interest; the notation $pos[\omega_j]$ refers to this position for ω_j. For example, in the SDFG of Figure 1, $pos[\omega_6] = 5$ since the relative position of ω_6 with respect to the beginning of the schedule of interest (i.e., s_1) is 5 (in Figure 7 consider 4 firings related to s_1 have been occurred before ω_6 and the 5^{th} actor firing related to s_1 is going to happen in ω_6). For each $\omega_j \in \Omega$ extracted for 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). 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 technique to detect all decision states within PSOS s_c. Assume s_c is a PSOS for the actors mapped to processor P_c. Schedules $s_{o1} \cdots s_{on}$ are PSOSs for the other actors of the SDFG mapped to the other processors (denoted by $P_{o1} \cdots P_{on}$). The output of Algorithm 2 is a set that contains all decision states for PSOS s_c. This algorithm also returns the relative positions of decision states with respect to the beginning of s_c. 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 running example, μ_0 and μ_1 are 1).

After normalizing the input schedules, all situations that can lead to multiple actors (mapped to the same processor) being ready to fire must be discovered. 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 by another actor in s_c. Processors can run at different clock rates; these differences and inter-processor dependencies cause variation in the number of tokens on the *inter-processor channels* originating from the actors mapped to the other processors to the actors mapped to the processor of interest (i.e., P_c). The number of tokens on the input channels of an actor determines whether an actor is enabled or not. To determine any possible actor enabling within s_c, a *necessary and sufficient number of tokens* on all inter-processor channels entering to the actors mapped to processor P_c must be considered. We will now explain what necessary and sufficient number of tokens means in our algorithm. Each iteration of the schedule of interest s_c requires that the actors mapped to 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 a *necessary number of tokens* on inter-processor channels entering to the actors mapped to processor P_c. More than one iteration for the other schedules $s_{o1} \cdots s_{on}$ may be feasible; this may cause an actor in s_c to be enabled more than its count in one iteration of s_c. The inter-iteration prevention constructs introduced in Section V-D control this undesired actor enabling. So, we only extract decision states within one iteration of the normalized schedule to provide a *sufficient number of tokens*.

Also, DSM does not impose any limitation between PSOSs since no dependency is created between actors mapped to different processors. 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 corresponding execution is named *maximal execution*. The maximal execution will stop at one point either due to a dependency on the actors on the processor P_c or because one PSOS iteration is completed. The SDFG state (denoted by ω_j in Algorithm 2) should be kept during the operation of the algorithm. The maximal execution is represented by the function *maxExec* in Algorithm 2. After one maximal execution, the number of tokens on the inter-processor channels entering into the actors on the processor P_c determines any possible enabled actor. The preserved state (i.e., 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 6 in Algorithm 2). The current position (i.e., i) in the schedule of interest s_c is also preserved for the discovered decision state (see line 5 in Algorithm 2). All enabled actors will be recorded as opponent actors of the state ω_j (line 7 in Algorithm 2). The execution of the actors on the processor

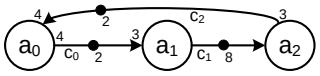


Fig. 8. A third example SDFG.

P_c is continued by executing the enabled actor in line with s_c to determine all possible decision states (line 8 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 process is repeated until a full iteration of the PSOS has been examined. In the end, the set Ω contains all possible decision states when executing s_c.

Figure 7 depicts the resulting state space from applying Algorithm 2 on the SDFG of Figure 1 when s_1 is the schedule of interest (i.e., s_c). In the SDFG of Figure 1, five consecutive decision states ($\Omega = \{\omega_5 \cdots \omega_9\}$) have been found for s_1 and no decision state has been found for s_0 (see Figure 7).

3) Redundant decision states: Here, we explain an optimization proposed in DSM to remove unnecessary decision states. DSM adds some components (per decision state) to create a dependency from the actor of choice of a decision state to the other opponent actors of that decision state. Such a dependency delays the firing of those opponent actors to the state after the decision state. The added components are explained in detail in Section V-E5.

It is possible to have several consecutive decision states which are delaying the firing of an actor to several states later. For example, three consecutive decision states ($\omega_7 - \omega_9$) exist in Figure 7 that all delay the firing of a_1; the added components in ω_7 delay the sixth firing of a_1 to ω_8; the added components in ω_8 delay the sixth firing of a_1 to ω_9; and so on. The latest decision state in the sequence of decision states $\omega_7 - \omega_9$ is enough to delay the firing of a_1 to 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 optimization can significantly reduce the number 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 Figure 1.

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 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 Figure 8. 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 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 since the same actor (i.e., a_2) is expected to fire in all of those states.

Modeling a repetitive behavior for a PSOS s_i, also models this behavior for its normalized PSOS (i.e., $s_i' = (s_i)^{\mu_i}$). Using this property, 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. Similarly, the 2^{nd}, 4^{th}, 6^{th} and 8^{th} states are grouped. So, the above execution shrinks to $\binom{a_2}{a_1}\binom{a_1}{a_2}$. After grouping all similar states in the original execution into a representative state, it is considered a decision state if any of the group members is a decision state. 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 ensures that only the actor of choice executes 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 firings related to those actors enabled in the last state except the actor of choice of that state are supposed to happen in subsequent PSOS iterations; this is already ensured by preventing inter-iteration execution (see Section V-D). Hence, after folding, the last state can be ignored as a decision state. In our third example, decision state folding reduces the number of decision states from 5 to 1 for the PSOS 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 Figure 7). Actor a_1 is the actor of choice in ω_6 and a_3 is the only opponent actor whose execution should be delayed to the state after ω_6. To enforce firing of a_1 and to prevent firing of a_3 in ω_6, DSM creates a dependency from a_1 to a_3 by adding actor $a_{1.\omega_6}$ and channels $c_{1.a_1\omega_6}$ and $c_{1.a_3\omega_6}$. The rates and initial tokens related to the new elements are set in such a way that the execution of the graph in other states are not affected. The actor $a_{1.\omega_6}$ is only responsible for decision state ω_6. So, $a_{1.\omega_6}$ needs to only fire once in an iteration. For this purpose, the port rates of $a_{1.\omega_6}$ on its channels (i.e., $c_{1.a_1\omega_6}$ and $c_{1.a_3\omega_6}$) are set to 6. The added dependency channels from the newly added actor in decision state ω_j (e.g., $a_{1.\omega_6}$ in ω_6) to the opponent actors which are not the actor of choice (e.g., a_3 in ω_6) only provide enough tokens for their execution in states $\omega_0 - \omega_{j-1}$ (e.g., 0 tokens for a_3 in $\omega_0 - \omega_5$); these actors cannot be enabled due to the lack of initial tokens in the newly added channels in the corresponding decision state (e.g., there will be no token in $c_{1.a_3\omega_6}$ in ω_6). Hence, only the actor of choice amongst the opponent actors of a decision state will be enabled in that state (e.g., only a_1 can fire in ω_6). The delayed actors in a decision state (e.g., ω_j) will have sufficient tokens on the incoming channel from the newly added actor for that 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 $c_{1.a_1\omega_6}$ after the firing of a_1 (i.e., the actor of choice) in decision state ω_6; hence, the actor $a_{1.\omega_6}$ immediately fires and then provides sufficient tokens for later firings of a_3. So, the delayed actor in decision state ω_6 (i.e., a_3) will no longer be blocked due to the absence of tokens in channel $c_{1.a_3\omega_6}$ after the decision state ω_6. The firing of actor a_1 after decision state ω_6 produces 1 token in channel $c_{1.a_1\omega_6}$ and the firings of actor a_3 after decision state ω_6 consumes 6

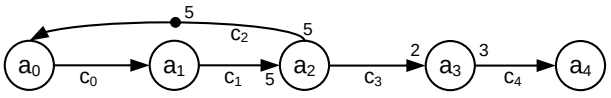


Fig. 9. An example SDFG.

Algorithm 3: SAS Modeling (SASM)

input : SDFG $G(A, C)$, PSOS $s_i = \{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$
output: G extended with schedule s_i

1 add a self edge with 1 initial token for each $a \in A_i$
2 **for** $i \leftarrow 1$ **to** n **do**
3 **if** α_i *is not an actor* **then**
4 SASM (G, α_i)
5 **AA**(G, a_{cnt_i})
 /* adding a counter channel */
6 **AC**(G, c_{cnt_i}, **rightMost**(α_i), a_{cnt_i}, 1, **RN**(**rightMost**(α_i), $\alpha_i^{\beta_i}$), 0)
 /* adding a limiter channel */
7 **if** $i = n$ **then**
8 **AC**(G, c_{lim_i}, a_{cnt_i}, **leftMost**(α_1), **RN**(**leftMost**(α_1), $\alpha_1^{\beta_1}$), 1, **RN**(**leftMost**(α_1), $\alpha_1^{\beta_1}$))
9 **else**
10 **AC**(G, c_{lim_i}, a_{cnt_i}, **leftMost**(α_{n+1}), **RN**(**leftMost**(α_{n+1}), $\alpha_{n+1}^{\beta_{n+1}}$), 1, 0)

tokens from channel $c_{1.a_3\omega_6}$; as a result, the number 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.

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 components added in decision state ω_9 show similar behavior as the components added in ω_6.

VI. Modeling Single Appearance Schedules

A well-known class of scheduling techniques are single appearance schedules (SASs) in which each actor appears exactly once in the LS form. This aspect makes SASs suitable to minimize code memory size. $s_2 = \langle(a_0a_1)^5a_2\rangle^*$ is a PSOS and SAS for part of the SDFG (i.e., actors a_0-a_2) in Figure 9.

DSM is able to model any PSOS. However, more compact graphs are possible for SASs. Algorithm 3 presents our single appearance schedule modeling (SASM) technique. Similar to DSM, SASM adds some extra actors and channels to the original SDFG to model the given schedules. The original channels and actors in the schedule-extended SDFG are preserved and distinguishable from the newly added elements by any of our techniques. Hence, both our techniques preserve the original structure of an SDFG. This property can be beneficial when a resource allocation algorithm needs to be applied on the schedule-extended graph; a resource allocation algorithm can easily distinguish an original actor (or channel) from an actor (or channel) which is added to model the schedules.

We know that each actor appears only once in a SAS; this property can help us to model a SAS in an SDFG in a smarter way than DSM does. An actor (or a nested inner LS) should be executed a specific number of times before another actor (or another nested inner LS) starts executing. In $s_2 = \langle(a_0a_1)^5a_2\rangle^*$, the nested inner LS (a_0a_1) must be executed 5 times before a_2 starts firing. This type of execution control can be handled using a counter construct. The key idea of SASM is to implement a counter concept in the graph. Later, we explain how we implement such counters to model SASs in an SDFG. Similar to DSM, auto-concurrency can be eliminated by adding a self-edge with 1 initial token to each actor in the SDFG (see line 1 in Algorithm 3). The rest of Algorithm 3 deals with implementing the counter concept.

Figure 10 shows the graph of the SDFG in Figure 9 which models the schedule s_2 using SASM. Schedule s_2 has a nested a_0a_1; this can be replaced with α_{01} to form a looped schedule representation (i.e. $s_2 = \langle(\alpha_{01})^5a_2\rangle^*$ where $\alpha_{01} = a_0a_1$). A counter in SASM is implemented by one actor a_{cnt_i} (e.g., a_{cnt_3} in Figure 10), a *counter channel* c_{cnt_i} (e.g., c_{cnt_3}) and a *limiter channel* c_{lim_i} (e.g., c_{lim_3}). A counter in SASM has two properties: first, it counts the number of times that element α_i (e.g., α_{01} above) is being executed; second, it limits the element α_{i+1} (e.g., a_2) to be executed to β_{i+1} times (e.g., 1 as the number of repetitions of a_2 is one). The counter channel c_{cnt_i} (e.g. c_{cnt_3}) is placed from the rightmost actor in α_i (e.g., actor a_1 in α_{01}) to the actor a_{cnt_i} (e.g., a_{cnt_3}); the production rate of the rightmost actor in α_i on c_{cnt_i} is set to 1 and the consumption rate of a_{cnt_i} on c_{cnt_i} is set to the number of times that the rightmost actor in α_i can be executed in ${\alpha_i}^{\beta_i}$ (e.g., 5 as the number of times that a_1 can be executed in ${\alpha_{01}}^5$ is five). In Algorithm 3, the function *rightMost*(α_i) (*leftMost*(α_i)) returns the rightmost (leftmost) actor in element α_i (e.g. *rightMost*($(a_0a_1)^5$) returns actor a_1). The function *RN*(a,${\alpha_i}^{\beta_i}$) retrieves the count of a in element ${\alpha_i}^{\beta_i}$ (e.g. *RN*(a_0,$(a_0a_1)^5$) returns 5).

The limiter channel c_{lim_i} (e.g., c_{lim_3}) is placed from a_{cnt_i} (e.g., a_{cnt_3}) to the leftmost actor in the next element, i.e., α_{i+1} (e.g., a_2). SASM performs the placement of the counter constructs in a circular way. In other words, the next element of α_j is considered to be $\alpha_{(j+1) \bmod n}$ where $j \in \mathbb{N} \wedge j \leq n$ for a LS $\{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$. The production rate of a_{cnt_i} on c_{lim_i} is set to the number of times that the leftmost actor in α_{i+1} can be executed in ${\alpha_{i+1}}^{\beta_{i+1}}$ (e.g., 1 as the number of times that a_2 can be executed in element a_2) and the consumption rate of the leftmost actor in the next element, i.e., α_{i+1}, from c_{lim_i} is set to 1. So, element α_{i+1} depends on actor a_{cnt_i} (because of c_{lim_i}) and actor a_{cnt_i} depends on α_i (because of the c_{cnt_i}); the β_i executions of α_i produce enough tokens on the counter channel c_{cnt_i} and then actor a_{cnt_i} can be fired. The firing of actor a_{cnt_i} provides enough tokens on limiter channel c_{lim_i} to only allow β_{i+1} executions for the next element α_{i+1}. Hence, by adding these components we enforce that α_{i+1} can be executed β_{i+1} times after α_i is executed β_i times.

The limiter channel of the counter construct added for the last element (i.e., $(\alpha_n)^{\beta_n}$) in a LS $\{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$ is initialized with some initial tokens to prevent a deadlock in the graph. The number of initial tokens on the limiter channel is set to the count of the left most actor of α_1 in the first element of that LS (i.e., $(\alpha_1)^{\beta_1}$). Line 8 in SASM performs the token initialization. Inter-iteration execution cannot happen because

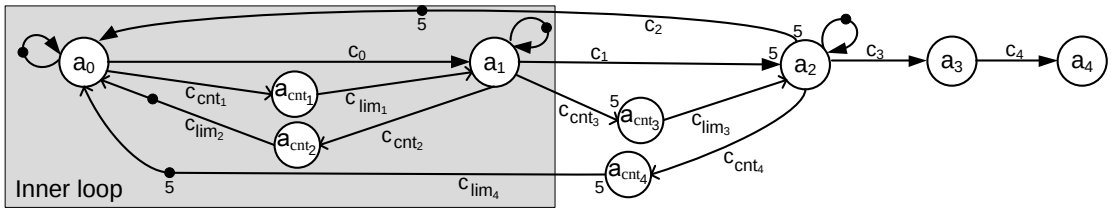


Fig. 10. SDFG of Figure 9 extended with $s_2 = \langle (a_0a_1)^5 a_2 \rangle^*$ using SASM.

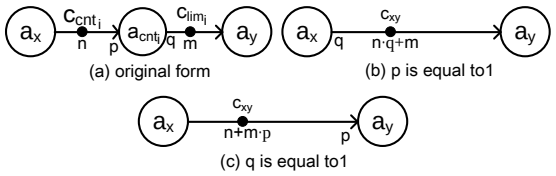


Fig. 11. Optimization of SASM.

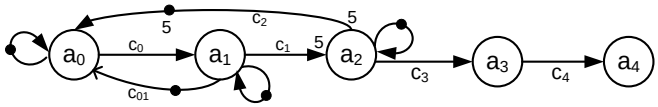


Fig. 12. SDFG of Figure 9 extended with $s_2 = \langle (a_0a_1)^5 a_2 \rangle^*$ using optimized SASM.

SASM always creates a dependency from the last actor in the schedule to the first actor in the schedule.

In Figure 10, the actor a_{cnt_3} is added to count the number of times that the sequence a_0a_1 is executed. The consumption rate of actor a_{cnt_3} on its input channel is 5; this means that after 5 executions of sequence a_0a_1 the next actor (i.e., a_2) can be enabled. Also, the actor a_{cnt_3} limits the number of times that actor a_2 should get enabled; this can be done by choosing the value 1 as production rate of actor a_{cnt_3} on its output channel. In other words, the actor a_2 can only fire once because of the limitation imposed by actor a_{cnt_3}. The actor a_{cnt_1} (a_{cnt_2}) is added to ensure the single execution of actors a_1 (a_0) after the single execution of actor a_0 (a_1). The actor a_{cnt_4} is added to ensure that the sequence a_0a_1 can be executed 5 times after the actor a_2 is executed once.

SASM is applied recursively (line 4 in SASM) to model the nested LS α_i. For example, SASM(G, a_0a_1) will be called inside SASM(G, $(a_0a_1)^5a_2$); the result of the recursive call is shown in Figure 10 with a rectangle marking the inner-loop.

Some of the elements added by SASM can be removed without affecting the outcome. Consider Figure 11(a) which contains a counter actor and two channels that can be discarded in the following cases:

- The counter actor a_{cnt_i} can be removed if rate p is equal to 1. The counter actor and two channels in the original form are replaced with channel c_{xy} (see Figure 11(b)).
- The counter actor a_{cnt_i} can be removed if rate q is equal to 1. The counter actor and two channels in the original form are replaced with channel c_{xy} (see Figure 11(c)).

The newly replaced channel c_{xy} is only necessary if there is no other equivalent channel in the original SDFG. The channels (p, q) and (p', q') which have the same source actor $a_x \in A$ and sink actor $a_y \in A$ are equivalent if the equation $\frac{Rate(p)}{Rate(p')} = \frac{Rate(q)}{Rate(q')} = \frac{\omega_0(c)}{\omega_0(c')}$ is true. Applying these optimizations on Figure 10 replaces all components added by SASM by channel c_{01} (see Figure 12). The SDFG which models schedule s_2 in the SDFG of Figure 9 with DSM and the HSDFG-based techniques result in a graph with 10 (26) and 13 (21) actors (channels) resp. The SDFG which models the same schedule with SASM only has 5 (9) actors (channels).

VII. Correctness of the Proposed Techniques

This section presents the theorems related to the correctness of our proposed techniques. A schedule is encoded correctly if any execution of a schedule-extended graph satisfies the encoded schedule and if any execution of the original graph that satisfies the given schedule is still possible in the schedule-extended graph. The proofs of the theorems are omitted for space reasons. They can be found in the report version of this paper [24].

Assume that the actors and channels added by DSM (or SASM) to model schedule s_i in SDFG $G(A, C)$ are denoted by A_{s_i} and C_{s_i}, resp. $G'(A', C')$ is the SDFG that models s_i in G using DSM (or SASM) where $A' = A \cup A_{s_i}$ and $C' = C \cup C_{s_i}$.

Theorem 1 and Theorem 2 state the correctness of DSM in modeling a single PSOS for a sub-set of the actors of the SDFG. Applying the theorems once for each schedule to be encoded shows the correctness of Algorithm 1.

Note that actors and channels added to model a given schedule do not affect actors that are not part of the schedule. Theorem 1 shows that any execution of the schedule-extended graph satisfies the modeled schedule. Firings of the added actors need to be ignored, which is achieved by the stated condition on the ordered lists resulting from the executions in the schedule-extended and original graphs.

Theorem 1. *Consider PSOS s_i as a schedule for actors $A_i \subseteq A$ from SDFG G (A, C). Assume $G'(A', C')$ is the SDFG that models s_i in G using DSM. 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 $orderList(\sigma, A) = orderList(\sigma', A)$.*

Theorem 2 shows that no execution of the original graph is unnecessarily excluded in the schedule-extended graph. In other words, any execution of the original graph that satisfies a given schedule is still feasible in the schedule-extended graph.

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

The following two theorems state the correctness of SASM. They are very similar to the two theorems for DSM.

Theorem 3. *Consider SAS $s_i = \{(\alpha_1)^{\beta_1}(\alpha_2)^{\beta_2}...(\alpha_n)^{\beta_n}\}$ as a schedule for actors $A_i \subseteq A$ from SDFG G (A, C). Assume $G'(A', C')$ is the SDFG that models s_i in G using SASM. 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 $orderList(\sigma, A) = orderList(\sigma', A)$.*

Theorem 4. *Consider PSOS s_i as a schedule for actors $A_i \subseteq A$ from SDFG $G(A, C)$. Assume $G'(A', C')$ is the SDFG that models s_i in G using DSM. For any execution σ of G that satisfies s_i it holds that there is exactly one σ' that is an execution of $G'(A', C')$ such that $orderList(\sigma, A) = orderList(\sigma', A)$.*

VIII. Experimental Results

In this section, we evaluate our techniques experimentally. We first explain the experimental setup. We then evaluate our techniques in terms of the sizes of the schedule-extended graphs, comparing our techniques to that of [12]. We further consider the throughput analysis time when analyzing the schedule-extended graphs obtained by different techniques. Note that the throughput that is achievable for a given schedule is independent of the way it is encoded. It is the analysis time itself that is of interest. Finally, we look at the accuracy of buffer sizing analysis. The accuracy of obtained buffer requirements does depend on the way schedules are encoded.

A. Experimental setup

The DSM and SASM techniques have been integrated in the SDF^3 [29] dataflow tool set, available at http://www.es.ele.tue.nl/sdf3. We use a set of DSP and multimedia applications (see the first column of Table I) to assess our DSM and SASM techniques.

In our experiments, applications are bound to a multi-processor platform using the technique of [6]. 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. We use a list scheduling approach from [30] to determine PSOSs for the applications. We use two different variations 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.

The scheduling technique presented in [31] is used to derive SASs for our benchmark applications. The technique in [31] also minimizes the required buffer sizes when determining a SAS. However, the technique in [31] cannot directly be used for multi-processors. We have utilized the technique of [31] to find SASs for a multi-processor platform. Initially the binding technique from [6] is used to bind the SDFG to a multi-processor platform. Then, the technique of [31] is applied to the SDFG to derive a SAS for all actors in the SDFG. This SAS is decomposed into some smaller SASs using the binding information; each of the smaller SASs is a schedule for one processor in the platform. Consider an example SDFG with 5 actors denoted by $a_0 - a_4$. Assume a_0, a_1 and a_3 are bound to processor P_1 and a_2 and a_4 are bound to processor P_2. Applying the technique of [31] to this imaginary SDFG gives the SAS $s_0 = \langle(a_0(a_1{}^2a_2a_3{}^4)^3)^2a_4{}^5\rangle^*$ for the whole SDFG. This SAS can be decomposed using the binding information to form a SAS for each of the processor in the platform. Only considering actors bound to P_1 in s_0 results in $s_{01} = \langle a_0(a_1{}^2a_3{}^4)^3\rangle^*$ which is a SAS for the actors bound to P_1. Similarly, a SAS $s_{02} = \langle a_2{}^6a_4{}^5\rangle^*$ can be extracted from s_0 to order actors bound to P_2. This way we utilize the technique of [31] for multi-processor platforms. The optimality of the generated schedules from the performance or buffer sizing perspective is debatable. However, we use this adapted SAS technique merely to provide some near-optimal inputs to evaluate our SASM schedule encoding technique versus the existing technique. Our techniques do not affect the quality of the scheduling result itself.

B. Comparison on graph sizes

Table I contains the size of the schedule-extended graphs using the HSDFG-based and DSM techniques to model Lfp and Lrp schedules for a single core platform (see the first two rows of each application line). Using schedules generated by Lfp, the number of decision states is less than when Lrp is used, except in the channel equalizer and MP3 playback 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 Figure 7) and the number of decision states can be reduced significantly. Usually SDFG actors closer to outputs are dependent on actors closer to inputs; this dependency can prevent an actor from being executed consecutively in a graph scheduled by Lrp. As a result, 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 decreases and extra elements are required to model the schedules in the graph. The exceptions in the channel equalizer and MP3 playback 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 number of decision states in DSM compared to Lrp. However, DSM always outperforms the HSDFG-based technique regardless of the input schedule in our experiments. The number of actors

TABLE I
SIZE OF THE SCHEDULE-EXTENDED SDFGS

Benchmark	Orig. size # actors (# channels)	Schedule type	# actors (# channels) HSDFG-based	DSM	SASM	Reduction compared to the HSDFG-based technique DSM	SASM
		Lfp	1190 (2973)	6 (14)	NA	99% (99%)	NA
H.263 dec. [10]	4 (6)	Lrp	1190 (2972)	598 (1198)	NA	49% (59%)	NA
		SAS	1190 (2972)	598 (1198)	4 (12)	49% (59%)	99% (99%)
		Lfp	201 (596)	7 (16)	NA	96% (97%)	NA
H.263 enc. [25]	5 (7)	Lrp	201 (499)	105 (212)	NA	47% (57%)	NA
		SAS	201 (499)	106 (214)	5(15)	47% (57%)	97% (97%)
		Lfp	911 (2849)	27 (61)	NA	97% (97%)	NA
MP3 dec. [10]	14 (21)	Lrp	911 (2327)	400 (807)	NA	56% (65%)	NA
		SAS	911 (2327)	400 (807)	14 (44)	56% (65%)	98% (98%)
		Lfp	48 (115)	22 (63)	NA	54% (45%)	NA
modem [1]	16 (35)	Lrp	48 (128)	35 (89)	NA	27% (30%)	NA
		SAS	48 (128)	35 (89)	16 (61)	27% (30%)	66% (52%)
		Lfp	612 (1639)	12 (29)	NA	98% (98%)	NA
samplerate conv. [1]	6 (11)	Lrp	612 (1784)	157 (319)	NA	74% (82%)	NA
		SAS	612 (1865)	238 (481)	12 (31)	61% (74%)	98% (98%)
		Lfp	4515 (11638)	41 (108)	NA	99% (99%)	NA
satellite rec. [26]	22 (48)	Lrp	4515 (12820)	1223 (2472)	NA	72% (80%)	NA
		SAS	4515 (15270)	3673 (7372)	24(91)	18% (51%)	99% (99%)
		Lfp	10601 (37531)	5298 (10600)	NA	50% (71%)	NA
MP3 playback [27]	4 (8)	Lrp	10601 (37529)	5296 (10596)	NA	50% (71%)	NA
		SAS	10601 (37530)	5297 (10598)	6 (16)	50% (71%)	99% (99%)
		Lfp	73 (341)	8 (20)	NA	89% (94%)	NA
bipartite [26]	4 (8)	Lrp	73 (359)	26 (56)	NA	64% (84%)	NA
		SAS	73 (350)	17(38)	9(22)	76% (89%)	87% (93%)
		Lfp	41 (100)	32 (83)	NA	22% (17%)	NA
channel eq. [28]	21 (40)	Lrp	41 (95)	27 (73)	NA	34% (23%)	NA
		SAS	41 (93)	30 (79)	21(65)	27% (15%)	48% (30%)

TABLE II
THE THROUGHPUT ANALYSIS TIME (IN MILLISECONDS)

		Type of the schedules / The used schedule modeling technique						
		SAS			Lfp		Lrp	
Benchmark	**# of processors**	**HSDFG-based**	**DSM**	**SASM**	**HSDFG-based**	**DSM**	**HSDFG-based**	**DSM**
H.263 dec.	2	36 540	< 1	< 1	118 720	< 1	120 710	< 1
H.263 enc.	2	380	< 1	< 1	690	< 1	400	< 1
MP3 dec.	3	13 900	320	10	17 660	< 1	13 980	1 380
modem	3	10	< 1	< 1	< 1	< 1	< 1	< 1
samplerate conv.	3	3 970	110	< 1	3 140	< 1	3 880	< 1
satellite rec.	2	not finished in 3 days	12 414 400	130	not finished in 3 days	280	not finished in 3 days	675 700
MP3 playback	2	not finished in 3 days	< 1	< 1	not finished in 3 days	10 780	not finished in 3 days	< 1
bipartite	2	30	< 1	< 1	40	< 1	50	< 1
channel eq.	3	< 1	< 1	< 1	< 1	< 1	< 1	< 1

(channels) using DSM is 66% (71%) lower compared to the HSDFG-based technique on average and 22% (17%) lower in the worst-case observed in our experiments. The average case refers to the mean value of the obtained results and the worst-case reports the smallest graph size reduction (i.e., reduction in numbers of actors and channels compared to the HSDFG-based technique).

SASs are a suitable class of schedules that minimize code memory size. DSM is able to model any arbitrary schedule in an SDFG. SAS can be modeled using DSM; however, it is possible to consider the intrinsic property of SASs when modeling a SAS in an SDFG. Our second technique, SASM, uses the fact that each actor appears only once in the looped schedule form. SASM models the counter concept in the graph in order to force actors to be executed a specific number of times. The third row of each application line in Table I contains the size of the schedule-extended graphs using the HSDFG-based, DSM and SASM techniques to model SASs, generated by the technique developed in [31].

SASM results in a schedule-extended SDFG with a limited number of extra actors and channels. For example, SASM only adds 2 (8) extra actors (channels) to the original graph of the MP3 playback application in order to model a SAS, while the HSDFG-based technique adds 1057 (37522) extra actors (channels) to model the same schedule. The graphs obtained by SASM have 88% (85%) and 48% (30%) less actors (channels) compared to the HSDFG-based technique on average and in the worst-case among the benchmark applications. Using the DSM technique to model the same SASs results in 46% (57%) and 27% (15%) less actors (channels) compared to the HSDFG-based technique on average and in the worst-case. Our results confirm that the techniques proposed in this paper achieve a more compact schedule-extended graph compared to the available technique.

C. Comparison on analysis times

The time required to perform an analysis on an SDFG depends on the size of the graph and the number of cycles in the graph. As an example, the throughput analysis of [9] is performed on the schedule-extended graphs using our techniques and the HSDFG-based technique. Our experiments are performed to evaluate the impact of the graph size on the analysis time of a common analysis technique. Note that other techniques (e.g., YTO [32]) can be employed to calculate throughput of HSDFGs, but the size of the schedule-extended

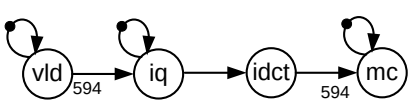
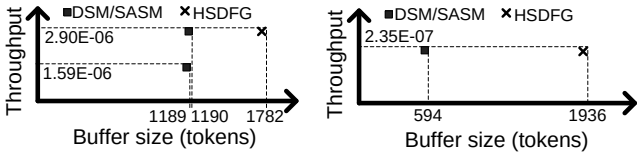


Fig. 13. SDFG of H.263 decoder application.



(a) H.263 decoder (b) MP3 decoder

Fig. 14. Pareto space of schedule-extended graphs modeled by DSM and HSDFG-based techniques (the scales of the two graphs are different).

HSDFGs is such that our conclusions remain the same. The benchmark graphs are mapped onto multi-processor platforms with two or three processors. Table II contains the throughput analysis times when SASs, list forward priority (Lfp) schedules and list reverse priority (Lrp) schedules are used as input schedules. The results show the superiority of SASM over DSM and the HSDFG-based technique. Note that the SDFG to HSDFG conversion is fast; the numbers reported for the HSDFG-based technique in Table II are related to the run-time of the throughput analysis from [9]. In our experiment, the run-time of a throughput calculation for HSDFGs is long independent of the analysis technique used (i.e., state-space [9], YTO [32], etc.).

D. Comparison on buffer sizes

To further analyze the effectiveness of our techniques, the buffer sizing algorithm from [10] is applied to the schedule-extended SDFGs of the H.263 decoder and MP3 decoder. Figure 13 depicts the SDFG of the H.263 decoder. Besides the compactness of the schedule-extended graph, our techniques preserve the original structure of an SDFG (when ignoring the added actors and channels), allowing accurate buffer sizing, which is not guaranteed for the state of the art technique. The H.263 decoder is mapped to a platform with two processors. The actors *vld* and *iq* are mapped to the first processor with a PSOS $\langle vld(iq)^{594}\rangle^*$ and the actor *idct* and *mc* are mapped to the second processor with a PSOS $\langle (idct)^{594}mc\rangle^*$. The analysis time for buffer sizing on the schedule-extended H.263 decoder is less than 1 ms when using DSM (or SASM) to model the schedules. The same analysis takes 1330 ms when using the technique from [12] to model the same schedules in the same graph. Figure 14(a) shows the Pareto space of throughput and buffer size when modeling the schedules with DSM (or SASM) and the HSDFG-based technique [12]. In this experiment, the schedules are first modeled in the graph; then, the buffer sizing technique of [10] is applied. 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 [12] results in 49% overestimation in required buffers compared to applying the same buffer sizing technique on the graph which models the schedules with one of our techniques. Note that the maximal achievable throughput is independent of the way schedules are encoded. The analysis results confirm this. Only the computed buffer sizes differ. For instance in both cases of Figure 14, the maximal throughput for the given schedules is always achievable, also by using the HSDFG-based schedule modeling technique; the latter suggests the need for larger buffers though. Figure 14(b) shows results for the MP3 decoder. We use the mapping and scheduling from [33] for a platform with 3 processors. The analysis time on the graph which models the schedule using one of our techniques is 594 ms while 141610 ms is required to perform the same analysis on the graph using the technique from [12]. Using the technique from [12] results in 226% overestimation in buffer size compared to using our techniques.

Modeling a PSOS in an SDFG using DSM requires execution of 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 DSM is below 1 ms. SASM also models SASs based on the structure of the schedules in their looped form; as each actor appears once in a SAS, the complexity of SASM depends on the number of actors in the graph. Similar to DSM, the execution time of SASM is always below 1 ms in our experiments. The complexity of our techniques relates to the length of the SDFG iteration and the number of processors in the platform (i.e., $|P|$). Hence, the complexity of our techniques is bounded to $O(|P| \cdot \sum_{a \in A} \gamma(a))$.

IX. Conclusion

We present two techniques, DSM and SASM, to model PSOSs and SASs 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 analysis techniques. Computing the trade-off between buffering and throughput for multi-processor platforms, for example, becomes several orders of magnitude faster. Moreover, properties like buffer sizes can be analyzed more accurately. The techniques have been integrated in the SDF^3 tool set available at http://www.es.ele.tue.nl/sdf3. This allows easy integration of the techniques in multi-processor design flows.

Acknowledgment

We thank the reviewers for their constructive comments, which helped improving the presentation of the paper. This work was supported in part by the Dutch Technology Foundation STW, project NEST 10346.

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 *et al.*, *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," in *DAC*, 2007, pp. 777–782.

[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 Transaction on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.

[11] M. Benazouz *et al.*, "A new method for minimizing buffer sizes for cyclo-static dataflow graphs," in *ESTIMedia'10*. IEEE, pp. 11 –20.

[12] 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.

[13] M. Damavandpeyma *et al.*, "Modeling static-order schedules in synchronous dataflow graphs," in *DATE'12*. ACM, pp. 775–780.

[14] A.-P. Wang *et al.*, "Buffer optimization and dispatching scheme for embedded systems with behavioral transparency," *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 4, pp. 41:1–41:26, Oct. 2012.

[15] M. Damavandpeyma *et al.*, "Throughput-constrained DVFS for scenario-aware dataflow graphs," in *RTAS'13*. IEEE.

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

[17] D. Thiele *et al.*, "Optimizing performance analysis for synchronous dataflow graphs with shared resources," in *DATE'12*. ACM, pp. 635–640.

[18] R. Henia *et al.*, "System level performance analysis - the SymTA/S approach," *IEE Proccedings Computers and Digital Techniques*, vol. 152, no. 2, pp. 148 – 166, mar 2005.

[19] L. Thiele *et al.*, "Real-time calculus for scheduling hard real-time systems," in *ISCAS'00*, vol. 4. IEEE, 2000, pp. 101 –104.

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

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

[22] E. Lee *et al.*, "Synchronous data flow," *Proceeding of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

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

[24] M. Damavandpeyma *et al.*, "Schedule-extended synchronous dataflow graph," TU Eindhoven, Tech. Rep., ESR-2013-01, 2013. [Online]. Available: http://www.es.ele.tue.nl/esreports/esr-2013-01.pdf

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

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

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

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

[29] S. Stuijk *et al.*, "SDF^3: SDF for free," ACSD. IEEE, 2006, pp. 276–278.

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

[31] P. K. Murthy *et al.*, "Joint minimization of code and data for synchronous dataflow programs," *Formal Methods in System Design*, vol. 11, no. 1, pp. 41–70, 1997.

[32] N. E. Young *et al.*, "Faster parametric shortest path and minimum balance algorithms," *CoRR*, vol. cs.DS/0205041, 2002.

[33] M. Geilen *et al.*, "Worst-case performance analysis of synchronous dataflow scenarios," CODES+ISSS. ACM, 2010, pp. 125–134.

Morteza Damavandpeyma received his B.Sc. and M.Sc. degrees in computer engineering from the University of Tehran, Iran in 2006 and 2008, respectively. He is currently a Ph.D. candidate in the Department of Electrical Engineering at the Eindhoven University of Technology (TU/e). His research interests include modeling, analysis and synthesis of embedded systems and multiprocessor System-on-Chips.

Sander Stuijk received his M.Sc. degree (with honors) in Electrical Engineering in 2002 and his Ph.D. degree in 2007 from the Eindhoven University of Technology. He is currently an assistant professor in the Department of Electrical Engineering at the Eindhoven University of Technology. His research interests include modeling methods and mapping techniques for the design, specification, analysis and synthesis of predictable hardware/software systems.

Twan Basten is a professor in the Electrical Engineering department at Eindhoven University of Technology (TU/e), the Netherlands, where he chairs the Electronic Systems group. He is also a senior research fellow of TNO Embedded Systems Innovation in the Netherlands. He holds an M.Sc. and a Ph.D. in Computing Science from TU/e. His research interests include embedded and cyber-physical systems, dependable computing and computational models.

Marc Geilen is an assistant professor in the Department of Electrical Engineering at Eindhoven University of Technology. He holds an M.Sc. in Information Technology and a Ph.D. from the Eindhoven University of Technology. His research interests include modeling, simulation and programming of multimedia systems, multiprocessor systems-on-chip, networked embedded systems and cyber-physical systems, and multi-objective optimization and trade-off analysis.

Henk Corporaal has gained an M.Sc. in Theoretical Physics from the University of Groningen, and a Ph.D. in Electrical Engineering, in the area of Computer Architecture, from Delft University of Technology. He is a professor in Embedded System Architectures at the Eindhoven University of Technology (TU/e) in The Netherlands. His current research projects are on low power single and multi-processor architectures, their programmability, and the predictable design of soft- and hard real-time systems.