Compositional Dataflow Modelling for Cyclo-Static Applications

Hadi Alizadeh Ara*, Marc Geilen*, Amir Behrouzian*, Twan Basten*†and Dip Goswami*
*Eindhoven University of Technology, Eindhoven, The Netherlands
†ESI, TNO, Eindhoven, The Netherlands

***Abstract*—Modular design is a common practice when designing complex applications for embedded systems. Another important practice in the embedded systems domain is the use of abstract models to realize predictable behaviour. Modular model-based design allows to construct a modular model of a complex system via model composition. The model of computation considered in this paper is scenario-aware dataflow, a dataflow model that allows for dynamic behaviour. We model applications with behaviour that changes according to a periodic pattern. Composing models with periodic patterns results in a model with a periodic pattern with a common hyper-period. We propose an efficient algorithmic method to compose cyclo-static scenario-aware dataflow models by generating composite patterns in a concise representation. We show that our approach can automatically generate concise models of several real-life image processing applications.**

I. INTRODUCTION

Complex real-time embedded systems are often developed using several design methods in the design process. Modular design is a design method that regards a system as several smaller, independently created modules that are interacting with each other through common interfaces. Modular design facilitates the creation of systems by composing pre-defined, reusable modules that can be designed separately. For instance, video encoder/decoders are typically constructed from a number of functional modules such as quantizers, transformers, sub-samplers, multiplexers, etc.

Model-based design is a mathematical method to deal with the design challenges of complex systems. In the embedded systems domain, this method is used to design a system that behaves according to an abstract model and consequently, has a predictable performance. Models are often supported by efficient performance analysis techniques. For instance, a number of timing models for real-time applications provide efficient techniques to accurately compute common timing performance metrics such as the throughput or latency. These methods provide performance guarantees and are often quicker to perform compared to the analysis performed on detailed implementations of the applications.

A complex real-time system with a predictable performance can be created with a modular model, constructed by composing the models of its modules. A composed model has the same type of interface as the modules have. This allows the composite models to be composed in an iterative manner to generate a hierarchical model of a complex system. A concise representation and scalable analysis methods for the model are very important, because the models can grow quickly.

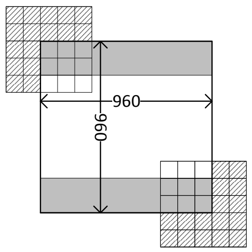


Fig. 1: A convolution filter

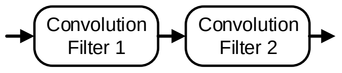


Fig. 2: Two producer-consumer composed convolution filters

Dataflow is a task-level Model of Computation (MoC) that expresses the behaviour of an application in terms of the execution dependencies between different tasks in the application. Synchronous Dataflow (SDF) [12], Cyclo-Static Dataflow (CSDF) [4] and Scenario-Aware Dataflow (SADF) [20] are different flavours of dataflow models that have been extensively used to design real-time embedded applications [18] [3] [11] [13], since they can be analysed for their structural and timing behaviours. SDF can model systems with behaviour that involves only execution dependencies that are static, i.e. they do not change during run-time. CSDF and SADF can express applications with behaviours that vary at run-time.

We focus on applications with behaviour that changes at run-time according to a known, periodic pattern. Such applications are often called cyclo-static. Cyclo-static applications are quite common in the real-time embedded system domain. Examples include audio applications that continuously alternate between left and right channel processing, video applications that use an alternation of horizontal and vertical filters or use multiplexers to periodically feed streams of data from several frequency bands into a single data stream for transmission.

CSDF and SADF are both capable of describing periodic behaviours (even though SADF is more expressive than CSDF). CSDF models are naturally modular, i.e., creating a CSDF model of an application from CSDF models of its modules is trivial. CSDF represents every module by a single directed graph and the modules can be trivially composed by connecting their graphs into a single composite graph. Although CSDF models are easy to compose, they lack analysis scalability and model compactness for applications with long periodic patterns. Since, cyclo-static application modules either may have long periodic patterns by themselves or are likely to create one when composed, they are not suitable for a modular model. For instance, consider a convolution filter operating on a stream of pixel data from a video with 960×960 frame size as shown in Figure 1. The filter uses a sliding window over the frames, for each frame starting from the top left pixel and sliding line by line from left to right. For every location of the window on the image, the filter requires that the pixels that fall under the window are available. Then it starts the convolution computations and finally produces an output pixel. Considering a stream of input pixels with the same arrival pattern as the movement of the window, the application repetitively goes through three different phases in terms of input-output pixel dependencies when processing each frame.

First, in the preparation phase, 2 input lines are consumed but no output is produced (the top gray area in Figure 1). Then, in the main phase, while the window moves forward pixel-by-pixel, input pixels are consumed and output pixels are produced. In this phase, 958 input lines are consumed (i.e. the remaining lines from an input frame) and the same number of output lines are produced. To produce output frames with the same frame size as the input frames, the filter goes to the termination phase where 2 output lines are produced (the bottom gray area in Figure 1) without consuming any inputs e.g. using a border padding technique. The frame-level behaviour of this application that captures input-output pixel dependencies throughout the frames, has a periodic pattern with period $2 + 958 + 2 = 962$ lines.

SADF specifies the behaviour of applications using a set of scenarios. A scenario models a sub-behaviour of the application, which includes a number of task executions with static dependency relations. To have a model that is quickest to analyse, the smallest (in terms of the number of task executions), different sub-behaviours are distinguished as separate scenarios. For example, the convolution filter can be described by a sequence composed of the following three scenarios: preparation scenario p in which one input line is consumed but no output pixels are produced, main scenario m that consumes a line of input pixels and produces a line of output pixels and, termination scenario t in which one output line is produced without consuming any input pixels.

A periodic behaviour is described by a repetitive sequence of scenarios in SADF, referred to as SADF sequences from now on. For instance, the frame-level behaviour of the filter can be described by a sequence where 2 repetitions of p are followed by 958 repetitions of m followed by 2 repetitions of t. Often SADF sequences contain a lot of repetitions. Therefore they can be compactly represented by an expression with a repetition construct. For example the sequence that describes the behaviour of the filter can be represented by expression $(p^2 m^{958} t^2)^\omega$ where ω denotes infinite repetition corresponding to an infinite stream of input frames.

Composing periodic behaviours results in a new periodic behaviour with a common hyper-period. For instance, consider the Producer-Consumer (PC) composition of two convolution filters as shown in Figure 2. The composite behaviour composes the different phases of both filters according to the data dependencies between two filters. In the first composite phase, when Filter 1 is in the preparation phase, Filter 2 is not active, since it does not have any inputs. In this phase, 2 input lines are consumed. In the second composite phase, while Filter 1 is producing outputs in the main phase, Filter 2 is in the preparation phase consuming the outputs produced by Filter 1. This phase consumes 2 input lines as well. In the next phase, Filter 2 goes to the main phase too and so on. This pattern repeats after the second filter completes its termination phase.

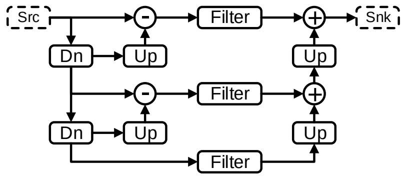


Fig. 3: A multi-resolution filter

The hyper-period can be described by a sequence of composite scenarios. A composite scenario describes a composite behaviour in which the dependency relations of the component scenarios are combined. For instance, a composite scenario denoted by $m \rightarrow p$ combines the dependency relations of scenario m of Filter 1 with scenario p of Filter 2, where the output pixels produced by Filter 1 are consumed by Filter 2. In this composite scenario, one input line is consumed (by scenario m in Filter 1), but no outputs are produced, since scenario p does not produce outputs. Using this composite scenario, the second phase in the hyper-period can be described by expression $(m \rightarrow p)^2$. To describe a composite phase in which one of the filters is not active, we can define a composite scenario where one of the scenarios is an empty scenario ϵ. For instance, the expression $(p \rightarrow \epsilon)^2$ can describe the first composite phase. By taking a similar approach we can describe the hyper-period using the following expression.

$$(p \rightarrow \epsilon)^2 (m \rightarrow p)^2 (m \rightarrow m)^{956} (t \rightarrow m)^2 (\epsilon \rightarrow t)^2$$

The hyper-periods of complex applications are often very long and computing them is not trivial. For instance, consider the 3-level multi-resolution filter shown in Figure 3. The image is fed to the lowest level. At each level, the image is decomposed into a difference image and a down-sampled image. The difference image is fed to a filter module and the down-sampled image is used as the input image of the next level. After filtering, the filtered images from all levels are added up to reconstruct the output image. This application has complicated data dependencies between different modules due to its non-trivial topology. This leads to the creation of many composite phases. Moreover, the filter, up-sampler and down-sampler modules have different periodic patterns. For instance, the down sampler pattern involves skipping the production of outputs for every other input line while the up sampler pattern shows the production of two lines per every input line. Composing modules with different periodic patterns results in a long hyper-period. Hence, the manual composition is tedious and error-prone. Therefore there is a need for an algorithmic method to automatically compute the composite behaviour.

We provide an approach to compose two cyclo-static modules given as SADF models. Our approach can be iteratively used to generate SADF models of applications that are composed of more than two modules. This approach involves two steps. First the hyper period of composition is computed. We provide an efficient algorithm for this purpose. The algorithm supports composition with producer-consumer and feedback synchronization on data dependencies between scenarios of different modules. The algorithm exploits the scenario repetitions in the scenario expressions to identify repetitive patterns in the composition. This technique is used to efficiently and directly generate a concise expression for the sequence of composite scenarios of the composition. The second step generates the $(\max, +)$ representations of all composite scenarios to perform timing analysis. This is done using the method of [17]. We show that our approach is fast enough to automatically generate concise models of several complex real-life image processing applications such as the multi-resolution filter.

II. Related Work

Compositionality of SDF models has been addressed in a number of works [6] [16] by introducing the concept of hierarchical SDF graphs. In a hierarchical SDF graph, a composite SDF actor is perceived as an atomic SDF actor. Such a representation is not compositional, i.e., it might lead to deadlock and/or rate inconsistencies [16]. The authors of [21] introduced a compositional abstraction for SDF actors, called Deterministic SDF with Shared FIFOs (DSSF) profiles. This allows for modular compilation and code generation for applications modelled as hierarchical SDF graphs.

In [17] a $(\max, +)$ algebraic throughput analysis method has been introduced for hierarchical SDF graphs. The approach exploits the inherent hierarchy in the graph when generating a $(\max, +)$ representation of the graph. This leads to a considerably faster analysis compared to the analysis performed on the unfolded graph. Another throughput analysis method for hierarchial SDF graphs is provided in [5]. The authors use the Interface-Based SDF (IBSDF) [15] as the composite actor model. This model extends the semantics of the SDF model to provide a graph composition mechanism based on hierarchical interfaces. An approximate throughput is obtained by constructing a periodic ASAP schedule for the IBSDF graph in a bottom-up approach.

Our work can be considered as an extension of the works above, since we allow for composite actors with cyclo-static nature. This adds the extra complexity of computing the hyper-period of the composite actor from the periods of its components. We use SADF to model the composite CSDF actors, where every scenario is represented by a $(\max, +)$ matrix. Generating the $(\max, +)$ representation of scenarios is done using the method of [17]. For conciseness, the periodic behaviours of the composite actors are expressed by ω-regular expressions that recognize scenario sequence words [1]. Computing the hyper-period of the composite actors which is the core part of this work, involves composing regular expressions in a specific way. We are not aware of other works that address this problem. We believe it may have interesting applications in other domains as well [14].

Another difference between this work and some of the works above is that the composite actors in this work communicate through channels that are not necessarily FIFOs. In fact the consumption order of the data on these channels is determined by the sequence of the consumer scenarios and not necessarily by the time the data is produced on the channels. This is necessary to ensure the correct timing of the cyclo-static models.

Other composable abstractions of dataflow models are based on actor interfaces [9]. These interfaces express the relation between sequences of input tokens and sequences of output tokens. The authors provide a composable refinement relation based on the earlier-is-better principle that preserves worst-case bounds on throughput and latency. [10] provides the Compositional Temporal Analysis (CTA) model based on these bounds. The authors show that this model can be used for buffer sizing and, it can accommodate latency constraints. In contrast to [9] and [10], we provide a composite model that has a dataflow behaviour as opposed to an abstraction of it.

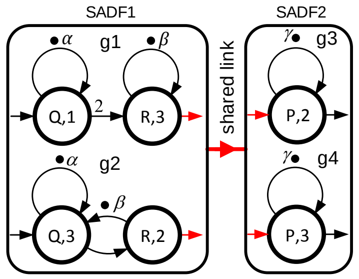
III. Preliminaries

A. *(max,+) Algebra*

$(\max, +)$ algebra supports two binary operators, namely maximum $(\oplus)$ and addition $(\otimes)$: $x \oplus y = \max\{x, y\}$ and $x \otimes y = x + y$. Scalars x and y belong to the union of real numbers and $-\infty$, which is denoted by $\mathbb{R}_{-\infty}$. Let vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ belong to the n dimensional vector space denoted by $\mathbb{R}^n_{-\infty}$. The inner product of $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined as $\boldsymbol{x}^T\boldsymbol{y} = (x_1 \otimes y_1) \oplus (x_2 \otimes y_2) \oplus \cdots \oplus (x_n \otimes y_n)$. Operator $\oplus$ on vectors with the same size, operates element-wise. Matrix multiplications and additions are defined using the above inner product and addition of vectors, similar to ordinary algebra. For readability we do not write $\otimes$ with matrix/vector multiplications.

B. *Synchronous Dataflow*

Actors and *channels* are the basic components of dataflow models. SDF specifies systems by a directed graph. There are four SDF graphs in Figure 4a (g1-g4). Graph nodes represent actors. Actors represent individual tasks or operations in the system. Directed edges represent channels. Channels form execution dependencies between tasks. We refer to actors by the letters inside the nodes. A channel from actor Q to actor R is denoted by (Q, R). Channels contain *tokens*. When an actor *fires* (executes), it consumes tokens from its inputs channels (incoming edges), takes a certain amount of time to execute and, finally produces tokens on its output channels (outgoing edges). Actor executions are *self-timed*. That is, actors execute as soon as their dependencies are met, i.e., when they have enough tokens on their input channels. The *execution times* of actors are denoted inside the nodes in the graphs. Channels may initially contain an integer number of tokens referred to as *initial tokens*. There are two initial tokens in g1, namely



(a) Composed SADF models

SADF	S	Seq.
SADF 1	a, b	$(ab^{10})^{\omega}$
SADF 2	c, d	$(cd)^{\omega}$

S	Graph
a	g1 (Q=1,R=2)
b	g2 (Q=1,R=1)
c	g3 (P=1)
d	g4 (P=1)

(b) SADF specifications

Fig. 4: An example of a producer-consumer composition

α and β. An SDF may be connected to an *open* channel i.e. a channel that is not connected to an actor on the other end. For instance graph g1 has an open input channel connected to actor Q and an open output channel connected to actor R. These channels can be used, for instance, to model streams of data that enter or exit the system or, to model execution dependencies between task executions in different graphs. We refer to tokens on the open input and output channels as *input tokens* and *output tokens* respectively. The number of tokens consumed from or produced on a channel by any firing of an SDF actor is constant and it is specified by the *rate* on the channel. Actor Q in g1 produces 2 tokens on its channel (Q, R) every time it fires. Rates of 1 are not shown in the graphs to avoid cluttering. SDF graphs are static models, since SDF actors have constant rates and execution times.

C. Scenario-aware Dataflow

SADF abstracts the behaviour of a system as sequences of static behaviours called *scenarios*. A scenario is a finite non-empty set of actor firings of an SDF graph. Figure 4b defines two SADF models: SADF 1 and SADF 2. Each of them has two scenarios; a and b for SADF 1 and, c and d for SADF 2. For instance, scenario a defines one firing for actor Q and two firings for actor R in graph g1. A scenario is said to be executed when all actor firings in the scenario are executed. *Final tokens* are tokens that are left on the channels after a scenario execution. For instance after an execution of scenario a, there is one token left on channel (R, R), one token on (Q, Q) and two (output) tokens on the open channel connected to actor R. A scenario labels its initial and final tokens in a number of channels, shown above the channels in Figure 4a. For instance α labels the initial and final tokens on channel (Q, Q) in the scenario graphs of a and b.

A long-run behaviour of an SADF can be obtained by consecutively executing scenarios. A particular periodic behaviour can be observed by repetitively executing a finite sequence of scenarios. Figure 4b specifies such a sequence for each SADF model. We use a compact representation for sequences called the repetition representation [14] [1]. For instance $(ab^{10})^{\omega}$ is an infinite scenario sequence made by indefinite repetition (ω) of the finite sequence a followed by ten repetitions of scenario b. When executing a sequence of scenarios, the execution dependencies among scenarios are established through the labelled initial and final tokens. When SADF 1 executes the scenario sequence ab, it executes scenario b after a, the production times of final tokens in scenario a become the availability times of initial tokens with the same labels in scenario b.

D. (max,+) Representation of Scenarios

Besides the graphical representation, dataflow models have a $(\max, +)$ representation. While the graphical representation makes it easier to distinguish different tasks and their dependencies in the system, the $(\max, +)$ representation abstracts the system into a set of critical dependency paths, making it more convenient for analysis. Moreover, we show that obtaining the $(\max, +)$ representation of composite scenarios can be done in a systematic and straightforward way in $(\max, +)$ algebra.

The dataflow semantics allows synchronization and delay. Consequently, the timing relations in dataflow is naturally captured by maximization and addition operations. For instance, consider the execution of actor Q in scenario a. Let t_α and t_ι denote the availability times of initial token α and an input token ι respectively and, t denote the production time of tokens produced by firing of Q. Actor Q starts its execution at $\max\{t_\alpha, t_\iota\}$ and completes it one time unit after the start. Therefore equation $t = \max\{t_\alpha, t_\iota\} + 1$ describes the relation between produced and consumed tokens in this firing.

The timing relation between the consumed and produced tokens caused by an actor firing can be expressed by a linear equation in $(\max, +)$ algebra. For instance, the equation above can be written as $t = 1 \otimes t_\alpha \oplus 1 \otimes t_\iota$. In a similar way, this algebra provides a unique representation for scenarios, sequences of scenarios and composite scenarios (discussed in the next section) through a system of linear equations in $(\max, +)$ algebra. These equations abstract the essential timing relations between the initial, final, input and output tokens [2].

Let $\boldsymbol{x}_s$ and $\boldsymbol{u}_s$ be vectors that contain the availability times of initial and input tokens in scenario s respectively. After s executes, the production times of final and output tokens are collected in vectors $\boldsymbol{x}'_s$ and $\boldsymbol{y}_s$ respectively. The $(\max, +)$ representation of a scenario s is described as follows.

$$\boldsymbol{x}'_s = \mathbf{A}_s \boldsymbol{x}_s \oplus \mathbf{B}_s \boldsymbol{u}_s \tag{1}$$

$$\boldsymbol{y}_s = \mathbf{C}_s \boldsymbol{x}_s \oplus \mathbf{D}_s \boldsymbol{u}_s \tag{2}$$

$\mathbf{A}_s$, $\mathbf{B}_s$, $\mathbf{C}_s$ and $\mathbf{D}_s$ are $(\max, +)$ matrices that describe the timing dependency relations in scenario s. Note that these equations are similar to state space representation of systems in control theory, where $\boldsymbol{x}$ denotes the internal states of the system and, $\boldsymbol{u}$ and $\boldsymbol{y}$ represent input and output signals respectively.

$(\max, +)$ matrices of a scenario can be computed using the symbolic simulation method provided in [7]. This method captures the timing relations of every actor firing using a vector dot-product in $(\max, +)$ algebra. Again consider the firing of actor Q in scenario a. The production time of tokens produced by this firing can be described as $t = \boldsymbol{g}^T \boldsymbol{t} =$

$[\ 1 \quad -\infty \quad 1\ ][\ t_\alpha \quad t_\beta \quad t_\iota\ ]^T$. Vector $\boldsymbol{t}$ contains the availability times of initial and input tokens in scenario a. Vector $\boldsymbol{g}$ is the symbolic time stamp vector assigned to every token produced by the firing of actor Q. The entry $-\infty$ indicates that there is no relation between the production time of the tokens and the availability time of β. As another example, we compute the time stamp of the first output token, $o1$. Token $o1$ is produced by the first firing of actor R, which consumes one of the tokens produced on channel(Q, R) with the time stamp of $[\ 1 \quad -\infty \quad 1\ ]$ and initial token β with the time stamp of $[\ -\infty \quad 0 \quad -\infty\ ]$. Therefore the time stamp of $o1$ is computed as $([\ 1 \quad -\infty \quad 1\ ] \oplus [\ -\infty \quad 0 \quad -\infty\ ]) \otimes 3 = [\ 1 \quad 0 \quad 1\ ] \otimes 3 = [\ 4 \quad 3 \quad 4\ ]$.

By symbolically simulating all actor firings in scenario a and collecting the time stamps of all final and output tokens we can obtain the following system of equations.

$$\begin{bmatrix} t'_\alpha \\ t'_\beta \\ t_{o1} \\ t_{o2} \end{bmatrix} = \left[\begin{array}{cc|c} 1 & -\infty & 1 \\ 7 & 6 & 7 \\ \hline 4 & 3 & 4 \\ 7 & 6 & 7 \end{array}\right] \begin{bmatrix} t_\alpha \\ t_\beta \\ t_\iota \end{bmatrix}$$

By letting $\boldsymbol{x}' = [\ t'_\alpha \quad t'_\beta\ ]$, $\boldsymbol{x} = [\ t_\alpha \quad t_\beta\ ]$, $\boldsymbol{u} = [\ t_\iota\ ]$ and $\boldsymbol{y} = [\ t_{o1} \quad t_{o2}\ ]$, the $(\max, +)$ matrices of scenario a are obtained as follows.

$$\mathbf{A}_a = \begin{bmatrix} 1 & -\infty \\ 7 & 6 \end{bmatrix} \mathbf{B}_a = \begin{bmatrix} 1 \\ 7 \end{bmatrix} \mathbf{C}_a = \begin{bmatrix} 4 & 3 \\ 7 & 6 \end{bmatrix} \mathbf{D}_a = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

The $(\max, +)$ representation of a finite sequence of scenarios can be obtained from the $(\max, +)$ representation of its scenarios. This is done by substituting the initial tokens vector $\boldsymbol{x}$ of the last scenario in the sequence with the final tokens vector $\boldsymbol{x}'$ of the one but last scenario in the sequence and, doing the same action for the rest of the scenarios in the sequence, all the way up to the first scenario. Finally by augmenting input and output matrices of all scenarios in the sequence, we can represent a scenario sequence in the form of Eqs. 1 and 2. The $(\max, +)$ representation of sequences will be used to compute the representation of the composite scenarios in the next section. In the rest of the paper, $\mathbf{A}_{\bar{s}}, \mathbf{B}_{\bar{s}}, \mathbf{C}_{\bar{s}}$ and $\mathbf{D}_{\bar{s}}$ denote the $(\max, +)$ matrices of a scenario sequence $\bar{s}$.

IV. Producer-Consumer Composition

This section defines the PC composition of two SADF models and provides an efficient algorithm to generate the composition. Figure 4 shows an example of this composition. The producer (on the left) and consumer (on the right) SADF graphs communicate through a *shared link*. That is, all scenarios in the producer SADF produce output tokens on the shared link and all scenarios in the consumer SADF consume input tokens from this link. To compose two SADF models we need to compose periodic sequences of the producer and consumer. The composition is a periodic sequence of *composite scenarios*. A composite scenario combines a finite sequence of scenarios from the producer with a finite sequence from the consumer such that its execution does not leave any tokens on the shared link. In other words, a composite scenario can be created from a non-empty pair of finite scenario sequences from the producer and consumer, such that the total number of outputs produced by the producer sequence matches the total number of tokens consumed by the consumer sequence. For instance, scenario a from SADF 1 can be composed with sequence cd from SADF 2, since a produces two outputs and c and d consume one input each.

A pair $(\bar{s}, \bar{t})$ of sequences is called a *composable pair* if the number of tokens produced by $\bar{s}$ is the same as the number of tokens consumed by $\bar{t}$. A composable pair can form a composite scenario. We denote the composite scenario by $\bar{s} \to \bar{t}$ and, we derive its $(\max, +)$ representation. When $\bar{s}$ and $\bar{t}$ are composed, the time stamps of the output tokens produced by $\bar{s}$ are assigned to input tokens consumed by $\bar{t}$ in the order determined by the sequences. For instance in the composite scenario $a \to cd$, the time stamp of the first output of a i.e. o_1 (note that it is logically the first, not necessarily temporally first) is assigned to the input of c and the second output, o_2 is assigned to the input of d. Since all inputs consumed by $\bar{t}$ are fed by outputs produced by $\bar{s}$, the inputs of the composite scenario are only the inputs of $\bar{s}$ and its outputs are only the outputs of $\bar{t}$ i.e. $\boldsymbol{u}_{\bar{s} \to \bar{t}} = \boldsymbol{u}_{\bar{s}}$ and $\boldsymbol{y}_{\bar{s} \to \bar{t}} = \boldsymbol{y}_{\bar{t}}$. The $(\max, +)$ representation of a composite scenario $\bar{s} \to \bar{t}$ can be obtained by substituting the input vector $\boldsymbol{u}_{\bar{t}}$ of $\bar{t}$ with output vector $\boldsymbol{y}_{\bar{s}}$ of $\bar{s}$ in the $(\max, +)$ representation of $\bar{t}$. Doing so and considering $\boldsymbol{x}_{\bar{s} \to \bar{t}} = [\ \boldsymbol{x}_{\bar{s}} \quad \boldsymbol{x}_{\bar{t}}\ ]^T$ and $\boldsymbol{x}'_{\bar{s} \to \bar{t}} = [\ \boldsymbol{x}'_{\bar{s}} \quad \boldsymbol{x}'_{\bar{t}}\ ]^T$ we can straightforwardly obtain the following $(\max, +)$ matrices for $\bar{s} \to \bar{t}$.

$$\mathbf{A}_{\bar{s} \to \bar{t}} = \begin{bmatrix} \mathbf{A}_{\bar{s}} & -\infty \\ \mathbf{B}_{\bar{t}}\mathbf{C}_{\bar{s}} & \mathbf{A}_{\bar{t}} \end{bmatrix} \qquad \mathbf{B}_{\bar{s} \to \bar{t}} = \begin{bmatrix} \mathbf{B}_{\bar{s}} \\ \mathbf{B}_{\bar{t}}\mathbf{D}_{\bar{s}} \end{bmatrix}$$

$$\mathbf{C}_{\bar{s} \to \bar{t}} = [\ \mathbf{D}_{\bar{t}}\mathbf{C}_{\bar{s}} \quad \mathbf{C}_{\bar{t}}\ ] \qquad \mathbf{D}_{\bar{s} \to \bar{t}} = \mathbf{D}_{\bar{t}}\mathbf{D}_{\bar{s}}$$

The hyper-period of periodic producer and consumer sequences is a sequence of composite scenarios that is composed of a whole number of periods of the producer and consumer sequences. The number of producer and consumer periods within the hyper-period depends on the total number of tokens produced by the producer and consumed by the consumer sequences in their periods. For instance, one period of $(ab^{10})^\omega$ produces 12 tokens and, 6 periods of $(cd)^\omega$ consume 12 tokens (2 tokens per period), therefore ab^{10} and $(cd)^6$ compose the hyper-period of the example in Figure 4. Trivially, the hyper-period can be a single composite scenario that is composed of the producer and consumer sequences in the hyper-period. For instance, we can create the composite scenario $(ab^{10} \to (cd)^6)$. However, such a composition hides the periodic patterns of the components inside possibly a gigantic scenario, which will have an adverse effect on the scalability of analysis. Moreover, such a gigantic composite scenario may have a very large $(\max, +)$ representation in terms of the sizes of matrices. For instance, if all phases of the filters in Figure 2 are composed into one scenario, the composite scenario will have millions of inputs and outputs for high resolution frames, which requires matrices $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ to have millions of rows and/or columns.

To preserve the periodic patterns of the producer and consumer in the hyper-period, we construct the hyper-period from composable pairs of sequences that are *minimal*. A composable pair of sequences is called minimal if no pair of prefixes from those sequences can make a composite scenario. For instance (ab, cdc) is not a minimal composite pair, because the prefixes a (from ab) and cd (from cdc) can form a composite scenario. A minimal pair of composable sequences makes a *minimal composite scenario*.

In a naive way, such a hyper-period can be obtained by successively taking minimal pairs of composable sequences from the producer and consumer sequences and composing them into a sequence of composite scenarios. For instance, let's consider the composition of sequences ab^{10} and $(cd)^6$. We start with composing a and cd and, create the first minimal composite scenario $a{\rightarrow}cd$. Now b^{10} is left from ab^{10} and $(cd)^5$ is left from $(cd)^6$. We proceed with composing b and c then, b and d and create minimal composite scenarios $b{\rightarrow}c$ and $b{\rightarrow}d$, which leaves us with b^8 and $(cd)^4$. By repeating b and c and, b and d composition four times more, we have completed the composition. This method produces a flat representation of the hyper-period, i.e. $(a{\rightarrow}cd)(b{\rightarrow}c)(b{\rightarrow}d)(b{\rightarrow}c)(b{\rightarrow}d)\cdots(b{\rightarrow}c)(b{\rightarrow}d)$. However, a compact representation $(a{\rightarrow}cd)((b{\rightarrow}c)(b{\rightarrow}d))^5$ is desired as the outcome.

The run-time of this naive approach scales linearly in the length of the hyper-period. For instance in SADF 1, if b repeats 10000 times instead of 10 times, the naive approach takes approximately 1000 fold more time to terminate, since b and c and, b and d composition will repeat for 5000 times in the hyper-period. Moreover, since it produces a flat representation of the hyper-period, it requires an additional step to obtain a concise representation from the flat representation. This additional step can be performed using the method introduced in [14]. This method has a time complexity of $\mathcal{O}(n^2 \log n)$ where n is the length of the flat representation. Therefore it dominates the complexity of the naive approach. We improve the scalability of the naive method by detecting repetitive patterns during the composition and directly generating a repetition representation of the composite sequence.

A repetitive pattern within the hyper-period occurs when repetitive subsequences from both producer and consumer sequences are composed. The repetitive pattern repeats until there are no repetitions left from at least one of the subsequences. Recognizing repetitive patterns during the composition can be done by tracking the remaining repetitions left from all repetitive subsequences of the producer and consumer sequences. To this end, we define a notion of *state* that carries the necessary information about the remaining repetitions from their subsequences. We use a representation for the states that is similar to the representation of the sequences, except that for the states, we use the notation n/m, where n is the initial number of repetitions and m is the number of remaining repetitions.

We illustrate the whole procedure by considering the composition of two convolution filters (Figure 2), as the filter sequences contain a lot of repetitions. Since all output pixels produced in one period of the producer filter are consumed in one period of the consumer filter, the hyper-period contains one complete period from each filter. Therefore, the consumer and producer sequences in the hyper-period

ALGORITHM 1: Hyper-period Computation

```
Input: The pair of sequences ($\bar{s}$,$\bar{t}$)
Output: The hyper-period sequence $\bar{r}$
1  $\bar{l} = \bar{r} = \epsilon$;
2  $(\tilde{s}, \tilde{t})$ = computeSequencesInHyperPeriod($\bar{s}, \bar{t}$);
3  $(S_p, S_c)$ = initialState($\tilde{s}, \tilde{t}$);
4  repeat
5      append $(S_p, S_c)$ to $\bar{l}$;
6      $\sigma$ = createMinimalCompositeScenario($S_p, S_c$);
7      append $\sigma$ to $\bar{r}$;
8      $(S'_p, S'_c)$ = newState($(S_p, S_c), \sigma$);
9      repeat
10         $\bar{\sigma} = \epsilon$; $k = 0$;
11         repeat
12             $(T_p, T_c) = l_k$;
13             if $S'_p \preceq T_p$ and $S'_c \preceq T_c$ then
14                 $\bar{\sigma}$ = getSequence($\bar{r}, (T_p, T_c), (S'_p, S'_c)$);
15                 $(i', i)$ = getRemainingReps($S'_p, T_p$);
16                 $(j', j)$ = getRemainingReps($S'_c, T_c$);
17                 break;
18             $k = k + 1$;
19         until $k \neq$ length($\bar{l}$);
20         if $\bar{\sigma} \neq \epsilon$ then
21             $r = \min\{\lfloor \frac{i}{i-i'} \rfloor, \lfloor \frac{j}{j-j'} \rfloor\}$;
22             replace $\bar{\sigma}$ with $\bar{\sigma}^r$ in $\bar{r}$;
23             $i'' = i - r(i - i')$; $j'' = j - r(j - j')$;
24             updateState($S'_p, i''$); updateState($S'_c, j''$);
25         $(S_p, S_c) = (S'_p, S'_c)$;
26     until $S_p = S_c = \epsilon$ or $\bar{\sigma} = \epsilon$;
27 until $S_p = S_c = \epsilon$;
```

are initially at state $(p^{2/0}m^{958/0}t^{2/0})^{1/1}$. This means that the whole producer/consumer filter sequence is yet to be composed, i.e., no scenarios from the filter sequence have been composed yet. We could also represent the initial state with $(p^{2/2}m^{958/958}t^{2/2})^{1/0}$. This representation can be obtained from $(p^{2/0}m^{958/0}t^{2/0})^{1/1}$ by reducing the remaining repetitions of the outer exponent by one and resetting the remaining repetitions of the inner exponents to their initial number of repetitions. The state representation in which the outer exponents have the highest number of remaining repetitions is called the *canonical representation*. For instance, $(p^{2/0}m^{958/0}t^{2/0})^{1/1}$ is the canonical representation of the initial state.

When a composite scenario sequence is created, the producer and/or the consumer states transition. Again consider the initial state $(p^{2/2}m^{958/958}t^{2/2})^{1/0}$ for both filters. When sequence $(p{\rightarrow}\epsilon)^2$ is composed (this sequence corresponds to the first composite phase), two p scenarios are taken from the producer sequence and the producer state transitions to $(p^{2/0}m^{958/958}t^{2/2})^{1/0}$. The consumer state does not change, because the composed sequence contains only empty scenarios from the consumer.

From the state transitions, we can identify repetitive patterns in the composite sequence. For instance, when the first $m \rightarrow p$ scenario is created, the producer state transitions from $(p^{2/0}m^{958/958}t^{2/2})^{1/0}$ to $(p^{2/0}m^{958/957}t^{2/2})^{1/0}$ and the consumer state transitions from $(p^{2/2}m^{958/958}t^{2/2})^{1/0}$ to $(p^{2/1}m^{958/958}t^{2/2})^{1/0}$. Comparing the states before and after this composition shows that, for the producer, the remaining repetitions of scenario m has been changed from 958 to 957 and, for the consumer, the remaining repetitions of scenario p has been changed from 2 to 1. Since for both producer and consumer there are differences in the remaining repetitions of exactly one exponent, that means repetitive subsequences have been composed and therefore, a repetitive pattern is detected. In this case the composite scenario $m{\rightarrow}p$ is repeated two times, since there are only two repetitions of p in the consumer sequence. Note that if we create the composite scenario $p^2m{\rightarrow}p$, the producer state transitions from $(p^{2/2}m^{958/958}t^{2/2})^{1/0}$ to $(p^{2/0}m^{958/957}t^{2/2})^{1/0}$. No repetitions of this composite scenario can be found in the hyper-period as there are differences in the remaining repetitions of two exponents.

In general, if after creating a composite sequence, the remaining repetitions of at most one exponent in the producer state changes from i to i' and the remaining repetitions of at most one exponent in the consumer sequence changes from j to j', a repetition of that composite sequence in the hyper-period is found and the number of repetitions is computed as follows.

$$r = \min\left\{ \left\lfloor \frac{i}{i-i'} \right\rfloor, \left\lfloor \frac{j}{j-j'} \right\rfloor \right\} \tag{3}$$

If the state of the producer/consumer transitions to itself (this happens when an empty scenario is composed into a composite scenario), the corresponding term is removed from Eq. 3.

After creating the repetitive pattern in the hyper-period the remaining repetitions of the producer and consumer subsequences are updated as follows.

$$i'' = i - r(i - i') \text{ and } j'' = j - r(j - j'). \tag{4}$$

We provide an efficient algorithm for the PC composition by detecting repetitive patterns via state comparison. Algorithm 1 sketches the method we propose for the composition. The algorithm accepts the repetition representation of the producer and consumer scenario sequences, $\bar{s}$ and $\bar{t}$ and, generates the composite sequence $\bar{r}$. The algorithm stores a sequence $\bar{l}$ that contains pairs of producer and consumer states. Initially $\bar{l}$ and $\bar{r}$ are empty sequences. The algorithm first computes the producer and consumer sequences in the hyper-period (line 2). Then, it obtains the pair of initial producer and consumer states and stores them in (S_p, S_c) (line 3). Lines 5-25 are repeated until the sequences in the hyper-period are completely composed. Line 5 appends the latest pair of producer and consumer states to $\bar{l}$. A new minimal composite scenario is created and appended to the composite sequence in lines 6 and 7. After the composite scenario is created, the new pair of states is stored in (S'_p, S'_c) (line 8). Then, it is compared with the pairs in $\bar{l}$, starting from the first pair i.e. the oldest pair (lines 12-18). An element of $\bar{l}$ is stored in (T_p, T_c). If T_p compares to S'_p and T_c compares to S'_c (we denote the comparison with operator $\preceq$ in the algorithm), then the

$\bar{l}$	S_p	S_c	$\bar{r}$
l_0	$(a^{1/1}b^{10/10})^{1/0}$	$(c^{1/0}d^{1/0})^{6/6}$	ϵ
l_1	$(a^{1/0}b^{10/10})^{1/0}$	$(c^{1/0}d^{1/0})^{6/5}$	$a{\rightarrow}cd$
l_2	$(a^{1/0}b^{10/9})^{1/0}$	$(c^{1/0}d^{1/1})^{6/5}$	$(a{\rightarrow}cd)(b{\rightarrow}c)$
l_3	$(a^{1/0}b^{10/8})^{1/0}$	$(c^{1/0}d^{1/0})^{6/4}$	$(a{\rightarrow}cd)(b{\rightarrow}c)(b{\rightarrow}d)$
l_4	$(a^{1/1}b^{10/0})^{1/0}$	$(c^{1/0}d^{1/0})^{6/0}$	$(a{\rightarrow}cd)((b{\rightarrow}c)(b{\rightarrow}d))^5$

TABLE I: Data of Algorithm 1 running on the example

sequence $\bar{\sigma}$ of minimal composite scenarios that are appended to $\bar{r}$ while transitioning from (T_p, T_c) to (S'_p, S'_c) is identified as a repetitive sequence. The remaining repetitions from the repetitive producer subsequence at states S_p and S'_p are stored in i and i', respectively. Similarly, j and j' respectively store these values for the consumer at states S_c and S'_c. If a repetitive pattern is detected, i.e., $\bar{\sigma}$ is non-empty, the repetition count is computed using Eq. 3, $\bar{\sigma}^r$ is created and replaced with $\bar{\sigma}$ in $\bar{r}$ and finally, the remaining repetitions i'' and j'' are computed using Eq. 4 and used to update S'_p and S'_c (lines 21-24). The algorithm goes to line 10 to check for another repetition after the states are updated. Otherwise the algorithm starts the next iteration (i.e. it goes to line 5) with the latest pair of states stored in (S_p, S_c) (line 25).

To illustrate, consider the example in Figure 4. The sequences ab^{10} and $(cd)^6$ are obtained as the producer and consumer sequences in the hyper-period. The initial pair of producer and consumer states is computed and appended to $\bar{l}$ (l_0 in Table I). Then the composite scenario $a \rightarrow cd$ is created and appended to $\bar{r}$. Now the latest pair of states, $((a^{1/0}b^{10/10})^{1/0}$, $(c^{1/0}d^{1/0})^{6/5})$, is compared with the first entry of $\bar{l}$ i.e. l_0. The comparison reveals a repetitive pattern, since in both producer and consumer, the remaining repetitions of only one exponent has been changed (exponent of a from $i = 1$ to $i' = 0$ and the outer exponent of consumer from $j = 6$ to $j' = 5$). Therefore, scenario $a \rightarrow cd$ repeats with the repetition count of one (using Eq. 3). Then the states are updated to $((a^{1/0}b^{10/10})^{1/0}$, $(c^{1/0}d^{1/0})^{6/5})$ using Eq. 4. The second iteration starts with appending the latest pair of states to $\bar{l}$ (i.e. entry l_1). Then the minimal composite scenario $b{\rightarrow}c$ is created. Comparing the pair of states after the creation of this scenario with l_0 does not show a repetitive pattern. However, comparing with l_1 shows a repetitive pattern on $b{\rightarrow}c$ with repetition of one. The flow of the third iteration is similar to the second iteration where $b{\rightarrow}d$ is created. In the fourth iteration, the comparison between the latest pair of states i.e. $l_3 = ((a^{1/0}b^{10/8})^{1/0}$, $(c^{1/0}d^{1/0})^{6/4})$ and the pair l_1 shows a repetitive pattern ($i = 10$, $i' = 8$, $j = 5$, $j' = 4$). Eq. 3 calculates the repetition of 5 for the sequence that is appended to $\bar{r}$ between the compared pairs of states, i.e. the sequence $(b \rightarrow c)(b \rightarrow d)$. This iteration is the last iteration, since both states become ϵ after they are updated using Eq 4. Therefore, the computed hyper-period is $(a{\rightarrow}cd)((b{\rightarrow}c)(b{\rightarrow}d))^5$.

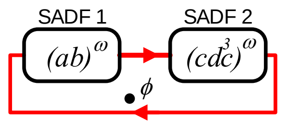
To explain the complexity of Algorithm 1, we assume the producer and consumer sequences are given by syntax trees and the hyper-period sequence is outputted as a syntax tree. The leaves of the syntax trees are labelled by scenarios and the inner nodes are either a binary operator labelled $\cdot$ (sequential composition), or they are a unary operator and labelled by ω or a natural number n (for the repetition). To compute the input sequences in the hyper-period (line 2) we need to compute the number of tokens produced and consumed by the producer and consumer in their periods, respectively. This can be computed by a reversed traversal of the input trees and it has the complexity $\mathcal{O}(n_{\bar{s}} + n_{\bar{t}})$ where $n_{\bar{s}}$ and $n_{\bar{t}}$ denote the sizes (the number of nodes) of the input syntax trees. In the iterative part of the algorithm (lines 5-25), by creating a new minimal composite scenario, a new leaf on the output tree is created. Also a new pair of states is added to list l which is looked up in every iteration. This means the algorithm scales quadratically in the number of leaves of the output tree. The number of leaves in the output tree is at least the maximum of the number of leaves in the input sequences. In the worst-case, when the input sequences do not match, in the sense that their composition does not produce any repetitive patterns (i.e. when the output turns out to be a flat sequence), the number of leaves in the output tree may grow as large as the length of the flattened hyper-period sequence.

It is worth mentioning that in the case of multiple consumers and producers, Algorithm 1 can be repetitively used to generate the composition. Moreover, the algorithm can handle the cases with more that one shared channel between the producer and consumer. In this case, the minimal composite scenarios are created such that their execution does not leave tokens on any of the shared channels.

V. Feedback Composition

This section generalizes the results of the previous section to the case where there are cyclic dependencies between component models. Figure 5a shows an example of such composition also known as feedback composition. Observe that the cyclic dependencies are established by two shared links in opposite directions. Initial token ϕ breaks the deadlock. SADF 1 has scenario sequence $(ab)^\omega$, where a and b are distinct scenarios. In SADF 2, c and d each denote a scenario and $(cd^3c)^\omega$ represents the sequence of scenarios. To compute the composite scenario sequence, we need to know the number of inputs and outputs consumed and produced by all scenarios. The table of Figure 5b specifies the number of inputs (I) and outputs (O) per scenario.

In the feedback composition, the composite scenarios are defined such that their execution does not change the number of tokens on the shared links. For example in Figure 5, scenario a and sequence cd^2 can create a composite scenario. We denote this scenario by $a{\leftrightarrow}cd^2$. The composite scenario $a{\leftrightarrow}cd^2$ can execute both scenario sequences in the composition and therefore, it is deadlock free. First, scenario a consumes token ϕ, executes, and produces two tokens on the link from SADF 1 to 2. At the same time, execution of c produces a token on the link from SADF 2 to 1. Then, two executions of d consume the tokens produced by scenario a. After these executions, the shared links contain the same number of tokens as they did before the execution of the composite scenario, which complies with the definition of the composite scenarios.



(a) Composed SADF models

S	a	b	c	d
I	1	1	0	1
O	2	1	1	0

(b) Scenario specifications

Fig. 5: An example of a feedback composition

Observe that this definition is consistent with the definition provided for the PC composition. In the PC composition, the composite scenarios do not leave tokens on the shared links because they are initially empty. Moreover, the PC composition cannot create deadlocks as it does not introduce any cyclic dependencies.

The $(\max, +)$ representation of composite scenarios in the feedback composition can be computed using the method provided in [17]. In this method, first the $(\max, +)$ representations of the components are generated by performing the traditional symbolic simulation [8] on the component graphs. Then, these representations are used in symbolic simulation of the composite graph to generate the $(\max, +)$ representation of the composite scenario. The deadlock freedom of the composite scenario is automatically checked during the simulation. That is, if there exists an actor firing schedule for all firings in the composite scenario, the simulation terminates and the composition is deadlock free.

Similar to the PC composition, the hyper-period combines the whole numbers of periods from the two components. Here, the hyper-period is defined only if the cyclic dependencies are correctly modelled, whereas the PC composition is guaranteed to have a hyper-period. For the feedback composition to have a hyper-period, the component periods must be consistent. That is, positive numbers p and q must exists such that the outputs produced by p periods of component 1 are consumed by q periods of component 2 and the outputs produced by q periods of component 2 are consumed by p periods of component 1. For instance, a period of SADF 1 consumes $1 + 1 = 2$ tokens and produces $2 + 1 = 3$ tokens. Similarly, one period of SADF 2 consumes $0 + 3 \times 1 + 0 = 3$ tokens and produces $1 + 3 \times 0 + 1 = 2$ tokens. Therefore the periods are consistent and have a hyper-period that contains one period from each component. As shown in the example above, the consistency of a feedback composition can be simply checked by solving a balance equation for every shared link, similar to the consistency check for SDF graphs.

Given a consistent pair of scenario sequences, Algorithm 1 can be used to compute the feedback composition, where lines 2 and 6 are performed considering the feedback composition rules. Using this algorithm, the hyper-period of the example shown in Figure 5 is computed as $(a{\leftrightarrow}cd^2)(b{\leftrightarrow}dc)$.

VI. Experimental Results

We implemented Algorithm 1 in SDF3 [19], a tool developed to analyse dataflow models and map them onto multiprocessor platforms. We used this tool to generate modular SADF models of a Multi-Resolution Convolution Filter

TABLE II: Analysis on SADF and CSDF models of MRCF

FS	L	T	SADF			CSDF	
			EL	CT	AT	NF	AT
960	3	0.2495	142	1	1	1.56e+7	1882
2048	3	0.2496	168	1	1	7.12e+7	8571
960	4	0.2495	186	1	1	1.72e+7	2070
2048	4	0.2497	210	2	3	7.86e+7	9228
960	5	0.2495	246	1	2	1.77e+7	2132
2048	5	0.2497	272	2	5	8.06e+7	9685

(MRCF) from the modular models of its up-sampler, down-sampler and filter components (see the structure of a multi-resolution filter in Figure 3), each of which is obtained by individually analysing the behaviour of their FPGA implementations for a given schedule and window size. We also used a throughput analysis [1] in this tool to compute the throughput of the generated models. The models are generated for square input images with different frame sizes (FS) and different numbers of layers (L) in the structure shown in Figure 3. Table II summarizes the experimental results. For each of these models, we report the throughput (T), in pixels per clock cycle, the hyper-period expression length (EL), in the number of characters used to represent the expression, the time needed to compose the models, i.e., composing time (CT), and the throughput analysis run-time (AT), both in milliseconds. The times are measured on an Ubuntu server with a 3.8Ghz processor. To compare the results with CSDF models, we generated CSDF graphs of this application by composing the CSDF graphs of the components. The CSDF analysis runs a dataflow simulation to compute the throughput. For this application, the simulation involves a huge number of actor firings (NF), which has an adverse effect on the analysis time. The results show that the computed throughput from both SADF and CSDF models are exactly the same for every configuration, and that the SADF models are quicker to analyze compared to CSDF models. Moreover, the modelling time is very short, even for such a complex application and, it is only slightly affected by the size of the image or the number of layers.

VII. Conclusion

We aimed to generate modular models of complex cyclo-static applications via model composition. For the sake of compactness and analysis scalability, we used the SADF MoC as the basic component model. We provided an approach to automatically generate SADF models of a cyclo-static application by composing SADF models of its components. The results showed that such models can be generated in a short time for complex applications. Moreover, the generated models can be quickly analysed compared to the traditional CSDF models.

VIII. Acknowledgment

This research is supported by the ARTEMIS joint undertaking through the ALMARVI project (621439) and ITEA 3 Project 14014 ASSUME.

References

[1] H. A. Ara, A. Behrouzian, M. Hendriks, M. Geilen, D. Goswami, and T. Basten. Scalable analysis for multi-scale dataflow models. *Accepted for publication in ACM Trans. on Embedded Computing Systems (TECS)*, 2018.

[2] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and linearity*, volume 2. Wiley New York, 1992.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2):151–166, Jun 1999.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. on signal processing*, 44(2):397–408, 1996.

[5] H. Deroui, K. Desnos, J. F. Nezan, and A. Munier-Kordon. Throughput evaluation of dsp applications based on hierarchical dataflow models. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017.

[6] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya. A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Proc. 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 189–198. ACM, 2008.

[7] M. Geilen. Synchronous dataflow scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, Jan. 2011.

[8] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proc. 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 125–134, New York, NY, USA, 2010. ACM.

[9] M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: A theory of timed actor interfaces. In *Proc. 14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 23–32, New York, NY, USA, 2011. ACM.

[10] J. P. Hausmans, S. J. Geuns, M. H. Wiggers, and M. J. Bekooij. Compositional temporal analysis model for incremental hard real-time system design. In *Proc. Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 185–194, New York, NY, USA, 2012. ACM.

[11] J. Keinert, H. Dutta, F. Hannig, C. Haubelt, and J. Teich. Model-based synthesis and optimization of static multi-rate image processing algorithms. In *Proc. 9th Conference on Design, Automation and Test in Europe*, DATE '09, pages 135–140, 2009.

[12] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, Sept 1987.

[13] O. Moreira and H. Corporaal. *Scheduling real-time streaming applications onto an embedded multiprocessor*, pages 67–75. Springer, 2014.

[14] A. Nakamura, T. Saito, I. Takigawa, M. Kudo, and H. Mamitsuka. Fast algorithms for finding a minimum repetition representation of strings and trees. *Discrete Applied Mathematics*, 161(10):1556 – 1575, 2013.

[15] J. Piat, S. S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *2009 IEEE Workshop on Signal Processing Systems*, pages 145–150, Oct 2009.

[16] J. L. Pino, S. S. Bhattacharyya, and E. Lee. A hierarchical multiprocessor scheduling system for dsp applications. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 122–126 vol.1, Oct 1995.

[17] M. Skelin and M. Geilen. Towards component-based (max,+) algebraic throughput analysis of hierarchical synchronous data flow models. In *International Conference on Computer Safety, Reliability, and Security*, pages 462–476. Springer, 2017.

[18] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proc. 44th Annual Design Automation Conference*, DAC '07, pages 777–782. ACM, 2007.

[19] S. Stuijk, M. Geilen, and T. Basten. Sdf3: Sdf for free. In *Proc. 6th International Conference on Application of Concurrency to System Design*, pages 276–278. IEEE, 2006.

[20] B. D. Theelen, M. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proc. 4th ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '06, pages 185–194. IEEE, 2006.

[21] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs. *ACM Trans. on Embedded Computing Systems (TECS)*, 12(3):83:1–83:26, 2013.