Parametrized dataflow scenarios

Technische Universiteit
Eindhoven
University of Technology

Parameterized Dataflow Scenarios

Mladen Skelin, Marc Geilen, Francky Catthoor, Sverre Hendseth

ES Reports
ISSN 1574-9517

ESR-2015-01
16 November 2015

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

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

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

Parameterized Dataflow Scenarios

Mladen Skelin[*], Marc Geilen[†], Francky Catthoor[‡], Sverre Hendseth[*]

Abstract

Although well-suited for capturing concurrency in streaming applications, purely dataflow-based models of computation are lacking in expressing intricate control requirements that many modern streaming applications have. Consequently, a number of modeling approaches combining dataflow and finite-state machines has been proposed. However, these FSM/dataflow hybrids struggle with capturing the fine-grained data-dependent dynamics of modern streaming applications.

In this article, we enrich the set of such FSM/dataflow hybrids with a novel formalism that uses parameterized dataflow as the concurrency model. We call the model FSM-based parameterized scenario-aware dataflow (PFSM-SADF). Through the use of parameterized dataflow, the formalism can capture the application fine-grained data-dependent dynamics while the enveloping FSM enables the capturing of the application control flow. We demonstrate the application of our modeling framework to synchronous dataflow (SDF), for which we propose a worst-case performance analysis framework based on the Max-plus algebraic semantics of SDF and the theory of Max-plus automata. We show that using the novel hybrid one can give tighter bounds on worst-case performance metrics such as throughput and latency for streaming applications exposing fine-grained dynamic behavior embedded inside a control-flow structure then by using the existing hybrids. We evaluate our approach on a realistic case-study from the multimedia domain.

1 Introduction and motivation

Dataflow models of computation (MoC) are widely used for modeling streaming applications typical examples of which can be found in the multimedia and signal processing domains. This is thanks to their simple graphical representation, compactness and the ability to expose parallelism contained in the considered application. Furthermore, the use of dataflow in a design process encourages good software engineering practices as modularity and code reuse. All the aforementioned, makes dataflow a natural design tool choice for streaming application designers.

A graphical representation of a dataflow MoC is a dataflow graph. In dataflow graphs, nodes are called *actors*, while edges are called *channels*. Actors represent computational kernels, while channels capture the flow of streams of data values between actors. These data values are called *tokens*. Tokens found in the graph before its execution commences form the set of *initial* tokens. Actors communicate by *firing*. Firing results in tokens being *consumed* from and *produced* in channels. We refer to token production and consumption numbers as *rates*. In timed dataflow, actor firing takes a finite amount of time called the actor *firing delay*.

With regard to expressiveness [37], dataflow MoCs can be roughly divided into two categories: static dataflow MoCs [24] and dynamic dataflow MoCs [7].

Most well-known examples of static dataflow MoCs are synchronous dataflow (SDF) [28], homogeneous SDF (HSDF), scalable SDF (SSDF) [33] and cyclo-static dataflow (CSDF) [8]. SDF is the most widely used dataflow MoC in general. In SDF, rates are constant and known at compile-time. HDF is a restriction of SDF composed of actors that only consume and produce one token per firing. SSDF is an extension of SDF where each actor may consume or produce any integer multiple of the

[1]Norwegian University of Science and Technology, Trondheim, Norway. E-mail: mladen.skelin@itk.ntnu.no, sverre.hendseth@itk.ntnu.no

[2]Eindhoven University of Technology, Eindhoven, The Netherlands. E-mail: m.c.w.geilen@tue.nl

[3]IMEC vzw, Leuven, Belgium. E-mail: catthoor@imec.be

predefined rate in one firing. This feature leads to better utilization of a DSP's arithmetic pipeline in synthesized implementations as actors operate on tuples of data samples. In CSDF, rates can vary between actor firings as long as the variation complies to a certain type of a periodic pattern.

Static dataflow MoCs, due to their restricted semantics provide predictability, enable static-scheduling and are amenable to powerful optimization techniques. Nevertheless, modern streaming applications are becoming increasingly dynamic. The increasing level of application dynamics hampers the use of static dataflow MoCs in design, analysis and implementation of contemporary embedded systems. The latter resulted in the need for expressive power well above the one obtainable from static dataflow models. This need gave rise to the class of dataflow MoCs we call dynamic dataflow MoCs. These models are able to capture dynamic applications, i.e. applications whose computation and communication requirements vary over time. With dynamic dataflow MoCs, actor rates vary in ways that are not entirely predictable at compile-time [7].

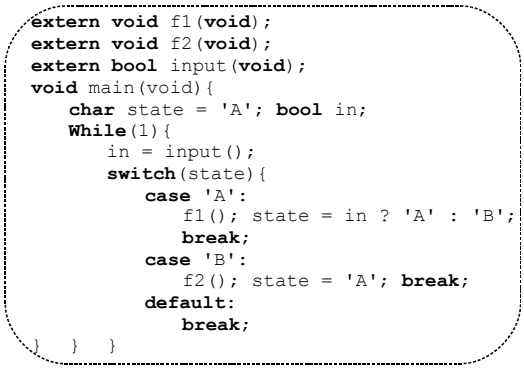
With regard to the concept used to represent the dataflow dynamics, the class of dynamic dataflow MoCs can be further refined into two subclasses [7].

The first subclass is composed out of dataflow MoCs that are developed around an interacting combination of finite-state machines (FSM) and dataflow graphs. Models such as heterochronous dataflow (HDF) [22] and FSM-based scenario-aware dataflow (FSM-SADF) [19] are well-known examples of such FSM/dataflow hybrids where an FSM is used to decouple control from concurrency. We bring further examples of such formalisms in Section 2.

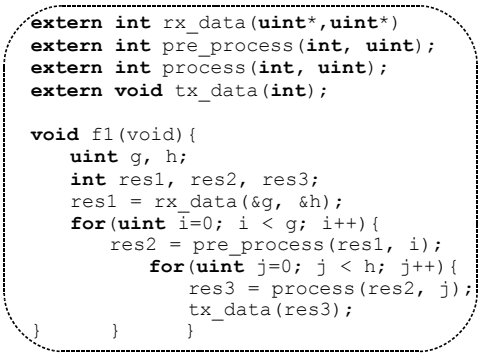
In the second subclass, dataflow dynamics are represented by alternative means [7]. In particular, the second subclass is formed by MoCs able to express complex data-dependent application dynamics without nourishing a particular state structure. Examples of such models are boolean dataflow (BDF) [10], dynamic dataflow (DDF) [10] and parameterized dataflow [7]. In BDF and DDF, data rates depend on the values of certain input tokens that are determined at run-time. In parameterized dataflow, the dataflow dynamics is represented by varying rates too, but by the use of dynamic parameters.

The dichotomy between the two subclasses lies in the notion of state. The MoCs of the first subclass maintain the notion of state, while the MoCs belonging to the second subclass do not. The need for the first subclass is justified by the existence of streaming applications with both intricate control requirements and concurrency [22]. The use of FSMs in expressing control requirements is justified by their finite nature and strong formal properties. The need for the second subclass stems from the fact that application's computation and communication requirements may depend on input data in very complex ways.

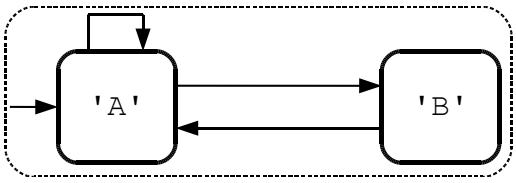
However, there exist many applications that both have intricate control requirements and are heavily input-data dependent in the sense that they may exhibit thousands or millions of behaviors depending solely on the characteristics of the input signal [25]. An example of such an application is shown in Fig. 1. The application is composed out of three modules: a control module and two data processing modules `f1` and `f2`. The C specification of the control module is shown in Fig. 1a, while its FSM specification is shown in Fig. 1c. The control structure is simple and involves transitions between states `'A'` and `'B'` depending on the current state and the value of the control input `in`. Within a state, the execution of a data processing module is invoked. C specification of one of the data modules, namely `f1` is shown in Fig. 1b. The module consists of two nested loops with bounds `g` and `h`. The loop bound values are input data-dependent as computed in the `rx_data` submodule. The actual implementation of the `rx_data` submodule involves complex input data processing. The derived bounds depend on some characteristics of the input signal. Assume that bound `g` can be assigned with a value originating from the interval $[0, m/2]$ while `h` can be assigned with a value from $[0, m/2]$. It this case, module `f1` will attain as many behaviors as there are integer points in the rational 2-polytope $P_{m,n}$ given by the set of constraints $\{0 \leq m/2, 0 \leq n/2\}$. With $n = 4500$ and $m = 2001$ the specification of Fig. 1b abstracts $2,252,126$ system behaviors [11]. Therefore, we can say that module `f1` and consequently the application as a whole expose fine-grained data-dependent dynamics recapitulated within the superordinate control structure. The data-dependent behavior of module `f1` can be succinctly expressed using the parameterized dataflow structure of Fig. 1d where loop bounds are abstracted into graph rates without the need for enumeration of all the rational 2-polytope $P_{m,n}$.



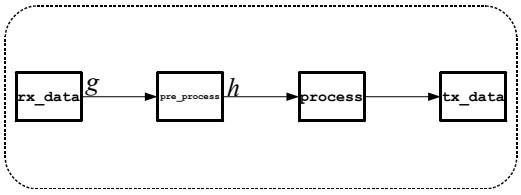
(a) C specification of the control module.



(b) C specification of `f_1` module.



(c) FSM specification of the control module.



(d) Dataflow specification of `f_1` module.

Figure 1: Motivational example.

For this type of dynamic applications, none of the MoCs belonging the two dynamic dataflow subclasses provide a succinct representation. The models from the first subclass can express the control structure of Fig. 1a through the notion of state, but cannot express fine-grained data-dependent dynamics of the subordinate data processing module of Fig. 1b. The models from the second subclass can express the latter, but not the former, i.e. their design interface cannot expose the control structure directly to the programmer [7]. Therefore, we argue that a combination of the two is needed.

In this article, we investigate an interacting combination of FSMs and parameterized dataflow. We use FSMs to capture the application control logic thanks to their finiteness, strong formal properties and an intuitive state abstraction that serves well for modeling control dominated parts of the application. We use parameterized dataflow to express fine-grained data-dependent dynamics of data-dominated parts of the application thanks to its ability to combine dynamic parameters and run-time adaptation of parameters in a structured way.

We base the novel model on the concept of scenarios adopted from [19]. Consequently, we model the execution of an application as a sequence of modes called scenarios, each of which is represented by a parameterized dataflow structure while the scenario occurrence patterns are given by the superordinate FSM. We refer to the novel model as FSM-based parameterized scenario-aware dataflow (PFSM-SADF). We demonstrate the application of PFSM-SADF concept to SDF as it is arguably the most used, mature and stable dataflow formalism. We refer to this specialization of PFSM-SADF as SDF-based PFSM-SADF (SDF-PFSM-SADF) for which we develop novel parametric worst-case performance analysis techniques based on the Max-plus algebraic [1] semantics of SDF and the theory of Max-plus automata [16].

The remainder of this paper is structured as follows. Section 2 gives an overview of existing parameterized dataflow models and FSM/dataflow hybrids. Section 3 introduces the preliminary concepts the crucial one being the Max-plus algebraic semantics of SDF and FSM-SADF as it will be of vital importance in developing the Max-plus semantics of SDF-PFSM-SADF. Section 4 presents our parameterized dataflow modeling framework. Section 5 presents the novel PFSM-SADF model. Section 6 defines the SDF-based specialization of PFSM-SADF and presents the Max-plus algebraic semantics of its constituents, i.e. SDF-based parameterized dataflow graphs (SDF-PDFG). Section 7

presents the Max-plus algebraic semantics and worst-case performance analysis techniques for SDF-PFSM-SADF. Section 8 demonstrates the application of our modeling and analysis framework using a VC-1 decoder case study. Section 9 concludes while setting directions to future work.

2 Related work

Parameterized dataflow as a meta-modeling technique was introduced by [6] and developed in the context of SDF yielding parameterized SDF (PSDF). PSDF specifications can be abstracted into actors in higher PSDF levels, which enables hierarchical integration. In PSDF, it is of vital importance, that the interface dataflow of a hierarchical actor remains unchanged throughout any iteration of its hierarchical parent actor. An iteration of a dataflow graph is a minimal nonempty set of actor firings that has no net-effect on the token distribution of the graph. This way, one maintains a level of predictability and permits efficient quasi-static scheduling at least for a class of PSDF specifications that satisfy certain technical constraints regarding the number of initial tokens placed on feedback channels.

Schedulable parametric dataflow (SPDF) introduced in [15], is a MoC closely related to PSDF. SPDF explicitly define requirements that a parameterized dataflow specification must satisfy so that questions about liveness (deadlock freedom), boundedness and schedulability can be answered at compile time. In contrast to SPDF, PSDF employs run-time mechanisms that check the consistency and bounded memory consistency of a specification. Boolean parametric dataflow (BPDF) [3] is a syntactical extension of SDF developed to elegantly treat cases where actor port rates may be zero. This is achieved by the introduction of conditional channels annotated with boolean expression. Depending on the value the expression attains at run-time, channel is to be activated or deactivated. Deactivation infers that no consumption or production can take place at that channel.

Variable-rate dataflow (VRDF) of [42] introduces facilities for frequent changes of actor port rates by means of parametrization. However, VRDF defines strong structural constraints that must be satisfied for achieving boundedness. More precisely, every production of p tokens must be coupled by exactly one consumption of p tokens. In addition, these pairs must be well-parenthesized in the graph. VPDF [41] is a CSDF-inspired generalization of VRDF where actors operate through sequences of phases and in each phase, the number of actor firings is parameterized along with the rates (or token transfer quantas in the parlance of [41]). The distinction is made to model loops for which no upper bound on their number of iterations is known. In contrast to VRDF, rates are allowed to have zero value and VPDF can model conditional execution.

Parameterized and interfaced dataflow meta-model or shortly PiMM [14] is obtained by enriching the meta-modeling techniques of [6] with the notion of interfaces as introduced in interface based synchronous dataflow (IBSDF) [31]. This ways, PiMM inherits the well-establish reconfiguration concepts of [6], while through the use of interfaces of IBSDF it defines notions to enable design reuse, i.e. the design of independent graphs that can be instantiated in an entirely different design layout.

To summarize, various flavour of parameterized dataflow models have been introduced as to support the growing need for efficient modeling tools that can capture both coarse and fine-grained reconfiguration phenomena present in modern streaming application. However, in their pure form, they are all inadequate for abstracting application control requirements as they do not depart from the dataflow framework and consequently do not provide interfaces to present application control requirements directly to the programmer. In addition, all of the models described expect VPDF and VRDF support no notion of time, i.e. they are untimed and therefore not accompanied by techniques for the analysis of their temporal behavior.

Next, we list models that do foster provision for expressing intricate application control logic directly to the programmer by defining precise semantics for integration of FSMs and dataflow.

Article [22] advocates the use of a combination of hierarchical state machines and various concurrent MoCs to decouple control from concurrency. The approach is referred to as *charts (pronounced *starcharts*). When SDF is used in conjunction with FSMs, the resulting model is called heterochronous dataflow (HDF). Two structural patterns are viable. First, an FSM can refine an

SDF actor. In this case, the FSM must obey the SDF semantics externally. Second, an SDF can be used to refine an FSM state. Then, SDF actor type signature (the number of tokens consumed and produced on each input and output) changes can occur only at iteration boundaries. This is assured by not allowing the FSM components to change state until the last actor firing within an iteration had completed.

Scenario-aware dataflow (SADF) MoC introduced in [38] enables modeling and analysis of dynamic systems by allowing actors to operate in different *modes* or *scenarios* across firings. In different scenarios, actors have different execution times and rates. SADF uses a stochastic approach to model scenario occurrence patterns. The operational semantics is defined in terms of a labeled transition system that can be analyzed to obtain both long-run average and worst-case performance metrics. SADF can be considered more expressive than HDF as it allows rates to change within an iteration.

FSM-SADF [19] is a model that is from the expressiveness point of view equivalent to HDF [37]. In HDF, each FSM state is mode-refined by a submodel, where each refinement has different rates while in FSM-SADF each FSM state is associated with an SDF model of a scenario the state corresponds to. Unlike HDF, which disallows for pipelined execution (it enforces purely sequential schedules) in FSM-SADF multiple scenarios can be active at the same time. FSM-SADF is a restriction of SADF in the sense that with FSM-SADF, scenarios can change only between complete iterations of SDF models of the respective scenarios, while with SADF, scenario changes are allowed within an iteration of the graph. Furthermore, the Markov automata of SADF is restricted to a (nondeterministic) FSM is FSM-SADF. On the other hand, FSM-SADF extends SADF because it allows auto-concurrent actor firings. The overall reduction in expressive power compared to SADF is advantageous from the analysis perspective. In particular, in FSM-SADF a clearer separation between nondeterministic control flow and determinate dataflow computations can be made and thus the associated max-plus spectral analysis-based algorithms for determining performance metrics avoid the state space explosion problems that the original SADF analysis is prone to.

The DF* (pronounced "DFstar") modeling framework of [13] is another dataflow MoC in the family of FSM/dataflow hybrids. A DF* graph is a network of blocks where each block consists of a set of code segments and a block controller. Each code segment specifies an alternative behavior of the block. The block controller is captured by an nondeterministic FSM. DF* is similar to HDF and consequently to FSM-SADF in the sense that code segments correspond to actor type signature.

Similar to other FSM/dataflow hybrids, the FunState MoC introduced in [39] defines precise semantics for separating dataflow from control in terms of functions driven by state machines.

Article [30] adds control flow provisions to bounded dynamic dataflow (BDDF) introduced in the same work. BDDF is an extension of SSDF allowing varying port rates with a requirement that the upper bound of each data rate must be specified. The control flow is specified as an FSM. Each state is defined by a network of blocks and it is executed repeatedly until a combination of multiple events causes it to be stopped in a non-preemptive manner and another state entered.

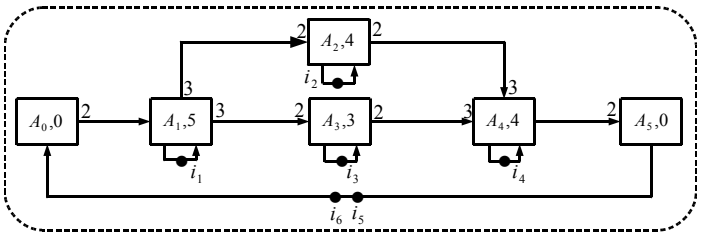
The modeling and simulation framework called El Greco [9] provides facilities to dynamically change specification parameters. It supports specifications given as combinations of dataflow graphs and hierarchical FSMs. Data-dependant dynamics can be captured using the limited support for parameterized data rates. However, the framework takes a more implementation oriented approach and is tailored for rapid simulation-based algorithm exploration. Therefore, it is not clear from [9] how static analysis is performed in the presence of parameters.

All the aforementioned hybrids are inadequate for representing fine-grained dataflow dynamics of data-dominated application modules as the number of behaviors they can express is typically proportional to the number of FSM states. Furthermore, all of them except SADF and FSM-SADF are untimed and thus not accompanied by corresponding temporal behavior analysis techniques.

3 Preliminaries

3.1 Max-plus algebra

We briefly introduce basic Max-plus algebra notation. Define $\mathbb{R}_{\max} = \mathbb{R} \cup \{-\infty\}$, where $\mathbb{R}$ is the set of real numbers. Let $a \oplus b = \max(a, b)$ and $a \otimes b = a + b$ for $a, b \in \mathbb{R}_{\max}$. For $a \in \mathbb{R}_{\max}$,



(a) Example SDFG.

(b) Schedule of the SDFG.

Figure 2: SDF.

$-\infty \oplus a = a \oplus -\infty = a$ and $a \otimes -\infty = -\infty \otimes a = -\infty$, i.e. $-\infty$ is the zero element of the $\oplus$ operation. By Max-plus algebra we understand the analogue of linear algebra developed for the pair of operations $(\oplus, \otimes)$ extended to matrices and vectors and denoted by $\mathcal{R}_{\max} = \{\mathbb{R}_{\max}, \oplus, \otimes\}$. The set of n dimensional Max-plus vectors is denoted $\mathbb{R}^n_{\max}$, while $\mathbb{R}^{n\times n}_{\max}$ denotes the set of $n \times n$ Max-plus matrices. The sum of matrices A, $B \in \mathbb{R}^{n\times n}_{\max}$, denoted by $A \oplus B$ is defined by $[A \oplus B]_{ij} = a_{ij} \oplus b_{ij}$ while the matrix product $A \otimes B$ is defined by $[A \otimes B]_{ij} = \bigoplus_{k=1}^{n} a_{ik} \otimes b_{kj}$. For a vector a and scalar c we use $c \otimes a$ or $a \otimes c$ to denote a vector with entries identical to entries of a with c added to each of them, i.e. $c \otimes a = a \otimes c = [a_i + c]$. For a vector a, $||a||$ denotes the vector norm, defined as $||a|| = \bigoplus_{i=1}^{n} a_i$. For a vector a with $||a|| > -\infty$, we use a^{norm} to denote $a - ||a|| = [a_i - ||a||]$, i.e. the normalized vector a. With $A \in \mathbb{R}^{n\times n}_{\max}$ and $c \in \mathbb{R}$, we use denotations $A \otimes c$ or $c \otimes A$ for $[a_{ij} + c]$. The $\otimes$ symbol in the exponent indicates a matrix power in Max-plus algebra. For $A \in \mathbb{R}^{n\times n}_{\max}$, $A^{\otimes k} = \bigotimes_{k} A$. For scalars c and α, $c^{\otimes\alpha} = \alpha \cdot c$.

3.2 Synchronous dataflow

This article investigates scenarios of parameterized dataflow graphs (PDFGs) later applied to SDF. Therefore, it is first needed to introduce the basic concepts of SDF. SDF as introduced in [28] is the most widely used dataflow MoC. Thanks to its static nature it is compile-time analyzable and characterized by large optimization potential. Fig. 2a shows an example of an SDF graph (SDFG) with six actors, while Fig. 2b specifies the graph's static schedule. Actors are depicted by rectangles while the production and consumption rates are annotated next to channel ends. If the value is omitted, a rate value of 1 is assumed. In SDF all input channels must contain enough tokens for the actor to fire. Every firing results in consumption and production of a fixed amount of tokens that is known at compile-time. Actor firing delays are denoted alongside actors names. Because of monotonic temporal behavior of SDF [18], one typically uses the worst-case execution times of code blocks that implement the actor's functionality for the actor firing delay. These can be obtained using state-of-the-art static timing analysis tools [43]. Existence of feedback loops in SDFGs will cause deadlock unless initial tokens (black dots in the figure) are appropriately placed on graph channels forming feedback loops. In SDF, initial tokens can be thought of as initial conditions for the execution rather than a part of the execution itself [32]. It is the schedulers responsibility to assure that all initial tokens are in place before the execution starts. We formally define an SDFG in Definition 1.

Definition 1 (SDFG)**.** *An SDFG $G = (\mathcal{A}, C, d, r, i)$ is a tuple where $\mathcal{A}$ is the set of actors, $C \subseteq \mathcal{A}\times\mathcal{A}$ is the set of channels, $d : \mathcal{A} \to \mathbb{R}_{\geq 0}$ returns for each actor its associated firing delay, $r : \mathcal{A}\times C \to \mathbb{N}_{>0}$ returns for each actor port its associated rate and $i : C \to \mathbb{N}_0$ returns for each channel its number of initial tokens.*

SDFGs can be scheduled at compile-time and thus implemented with minimal run-time overhead. Schedule for an SDFGs is a loop over a series of actor firings completing an iteration. Recall that,

an iteration of a dataflow graph is a minimal nonempty set of actor firings that has no net-effect on the token distribution of the graph. The schedule for the running example SDFG is shown in Fig. 2b. The same schedule can be denoted via the term $A_1^1 A_2^2 A_3^3 A_4^3 A_5^2 A_6^1$ where exponents represent actor repetition counts. We consider SDFGs that are consistent and deadlock-free. The graph that is inconsistent may deadlock or be unbounded which means that it has no unbounded execution with bounded buffers [32]. Paper [28] shows how to compute the minimal numbers of times that an actor needs to be fired in a valid schedule by using the so called balance equations. The solutions to the balance equations are stored in the repetition vector of the graph which we define as a map $\Gamma : \mathcal{A} \to \mathbb{N}_{>0}$. With the abuse of notation, for the running example, $\Gamma(A_1, A_2, A_3, A_4, A_5, A_6) = (1, 2, 3, 3, 2, 1)$. The existence of the repetition vector implies consistency. However consistency does not imply that a valid schedule exists. If a graph contains cycles, it may deadlock although consistent. That is why sufficient numbers of initial tokens must be placed in feedback channels. Checking the deadlock-freedom of an SDFG is performed by computing an iteration by abstract execution [28].

3.3 Max-plus algebra for SDF

Max-plus algebra [1] is used to capture the semantics of self-timed execution od SDF. Self-timed execution is a schedule where every actor fires as soon as possible, i.e. immediately when all input channels contain a sufficient number of tokens. These numbers are specified by rates. The self-timed execution defines the tightest bound that can be given on the temporal behavior of the system captured by an SDF model [18]. In relation to Max-plus algebra, two fundamental concepts that determine the self-timed execution of an SDFG are *synchronization* and *delay*. Synchronization manifests itself when an actor waits for all input tokens to become available. The delay manifests itself through the fact that the tokens that are the result of an actor firing will be available after an amount of time following the firing start time. This amount of time is equal to the actor firing delay. The two concepts described correspond to the two operators of Max-plus. Synchronization corresponds to the max operator while delay corresponds to the + operator of Max-plus. To exemplify, we follow [18]. Let T be set of tokens an actor requires to start its firing. Let t_τ be the availability time of token $\tau \in T$. Let d be the firing delay of the actor. In this case the output tokens produced by the actor will become available at time

$$\max_{\tau \in T} t_\tau + e = \bigoplus_{\tau \in T} t_\tau \otimes e. \tag{1}$$

In (1) and in the remainder of this paper for max we use the symbol $\oplus$ (pronounced "o-plus") and for + we use the symbol $\otimes$ (pronounced "o-time").

SDFGs evolve in iterations. Therefore, the beginning and the end time of any SDFG iteration are fully determined by the availability times of initial tokens. Recall that initial tokens represent initial conditions for execution. If the production timestamps of initial tokens after the kth graph iteration are collected in the vector $\gamma(k)$ the evolution of an SDFG under self-timed scheduling policy is given by the following recursive Max-plus linear equation

$$\gamma(k+1) = M_G \otimes \gamma(k). \tag{2}$$

In (2), $M_G \in \mathbb{R}_{\max}^{|I| \times |I|}$ is the SDFG Max-plus matrix, I is the set of initial tokens of the SDFG and $\gamma(k)$ is the timestamp vector of the kth SDFG iteration. Matrix M_G is a square matrix, which follows from the fact that each initial token has one entry in $\gamma(k+1)$. For initial tokens, throughout this article, we use the notation i_l where $l \in \{1, \ldots, |I|\}$, so that l specifies the position of the initial token's timestamp in the timestamp vector and notation I for the set of graph's initial tokens.

To exemplify, consider the running example graph with $I = \{i_1, \ldots, i_6\}$. The initial vector is $\gamma(0) = [0, 0, 0, 0, 0, 0]^T$. After one iteration $\gamma(0)$ becomes $\gamma(1) = [10, 18, 16, 22, 22, 0]^T$; after two iterations $\gamma(2) = [20, 30, 26, 34, 34, 22]^T$ and so on. From the recursion of (2), we can derive an explicit function for $\gamma(k)$ as follows

$$\gamma(k) = {M_G}^{\otimes k} \otimes \gamma(0). \tag{3}$$

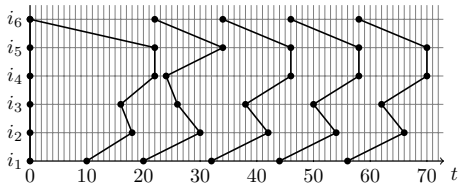


Figure 3: Execution of the SDFG of Fig. 2a.

Fig. 3 shows the evolution of the timestamp vector for the first five iterations of the running example SDFG.

Time is depicted horizontally and the six tokens of the timestamp vector are depicted vertically so that lines visualize the timestamp vectors of (2), i.e. (3). The sequence $\gamma(0), \gamma(1), \gamma(2), \ldots$ defined by (2), i.e. (3) is ultimately periodic so that the following recursive equation holds for all $k \geq k_0$

$$\gamma(k+\sigma) = \lambda^{\otimes\sigma} \otimes \gamma(k). \tag{4}$$

In (4), k_0 is the transient time of M_G, σ is the cyclicity of M_G and λ is the eigenvalue of M_G.

Moreover, (2) and (4) define an alternative iteration-based operational semantics of SDF in contrast to the traditional actor firing-based operational semantics of [20] that cannot separate iterations. The periodicity of SDF expressed via (4) is reflected in the existence of the periodic phase in the state-space of the SDFG constructed by the use of its traditional operational semantics. If we define throughput of an SDFG as the long-run average of completed iterations per time-unit [21], it straightforwardly follows (4) that the inverse of λ defines the throughput of the graph.

Matrix M_G of (2) can be derived by symbolically executing one iterations of the corresponding SDFG with the intention of relating the entries of $\gamma(k+1) = [t'_{i_1}, \ldots, t'_{i_{|I|}}]$ and $\gamma(k) = [t_{i_1}, \ldots, t_{i_{|I|}}]$ where t'_{i_l} and t_{i_l} are the timestamps of the corresponding initial tokens after the $(k+1)$st and the kth SDFG iteration embodied into the timestamp vectors of the $(k+1)$st and the kth iteration, respectively.

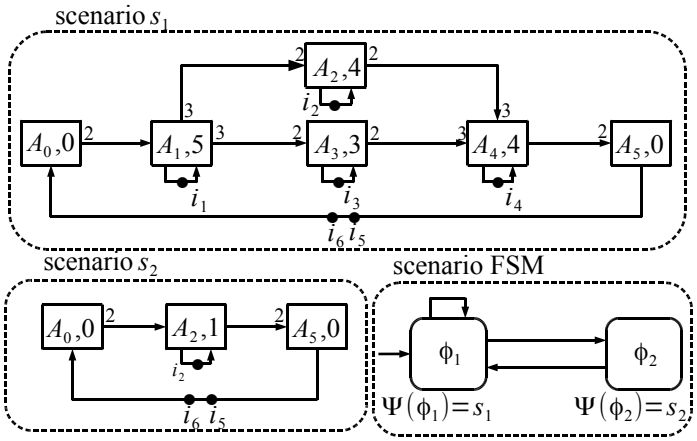
First, consider the following. It was shown in [18], that the production timestamp t of any token can be represented as a Max-plus scalar product

$$t = \bigoplus_{i_j \in I} m_j \otimes t_{i_j} = [m_1, \ldots, m_{|I|}] \otimes \gamma(k). \tag{5}$$

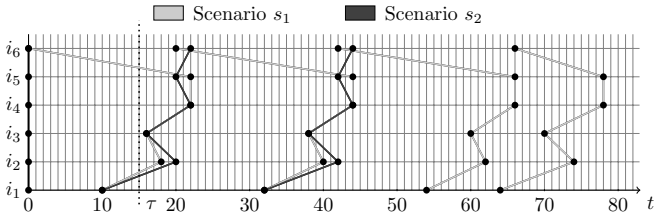
between a vector of suitable constants called the *initial token dependency vector* or shortly the dependency vector and the timestamp vector of the kth iteration. Then, also the entries of $\gamma(k+1)$ can be written as linear combinations of entries of $\gamma(k)$ as follows

$$t'_{i_l} = \bigoplus_{i_j \in I} m_{l,j} \otimes t_{i_j} = [m_{l,1}, \ldots m_{l,|I|}] \otimes \gamma(k). \tag{6}$$

It straightforwardly follows from (2) and (6) that dependency vectors $[m_{l,1}, \ldots m_{l,|I|}]$ define the rows of M_G. These vectors are determined by symbolic execution of one iteration of the graph. We show this with an example. Running example SDFG has six initial tokens. We represent the timestamp vector of the kth graph iteration as $\gamma(k) = [t_{i_1}, t_{i_2}, t_{i_3}, t_{i_4}, t_{i_5}, t_{i_6}]^T$. Similarly, the timestamp vector of the $(k+1)$st iteration is represented as $\gamma(k+1) = [t'_{i_1}, t'_{i_2}, t'_{i_3}, t'_{i_4}, t'_{i_5}, t'_{i_6}]^T$. According to the periodic schedule (cf. Fig. 2b) actor A_1 fires first. In order to fire, A_1 must consume token i_6. Therefore, the tokens produced by its firing are determined by the timestamp vector $t_{i_6} \otimes 0 = [-\infty, -\infty, -\infty, -\infty, -\infty, 0] \otimes \gamma(k)$. Note that we express timestamps t_{i_l} as Max-plus scalar products too. In this case, $t_{i_6} = [-\infty, -\infty, -\infty, -\infty, -\infty, 0] \otimes \gamma(k)$. Thereafter, actor A_2 fires two times. The first firing produces the tokens determined by the timestamp vector $([-\infty, -\infty, -\infty, -\infty, -\infty, 0] \otimes \gamma(k) \oplus [0, -\infty, -\infty, -\infty, -\infty, -\infty] \otimes \gamma(k)) \otimes 5 = [5, -\infty, -\infty, -\infty, -\infty, 5] \otimes \gamma(k)$. The second firing produces tokens carrying the timestamp vector $[10, -\infty, -\infty, -\infty, -\infty, 10] \otimes \gamma(k)$. Furthermore, the second firing of A_2 restores i_1 and therefore $t'_{i_1} = [10, -\infty, -\infty, -\infty, -\infty, 10] \otimes \gamma(k)$. By continuing



(a) Example FSM-SADFG.



(b) Execution of the example FSM-SADFG.

Figure 4: FSM-SADF.

the symbolic execution until the completion of the iteration we will obtain the new timestamps of remaining initial tokens. By collecting the corresponding dependency vectors, we obtain M_G of (7).

$$M_G = \begin{bmatrix} 10 & -\infty & -\infty & -\infty & -\infty & 10 \\ 18 & 12 & -\infty & -\infty & -\infty & 18 \\ 16 & -\infty & 9 & -\infty & -\infty & 16 \\ 22 & 16 & 14 & 8 & -\infty & 22 \\ 22 & 16 & 14 & 8 & -\infty & 22 \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \end{bmatrix} \tag{7}$$

3.4 Synchronous dataflow scenarios

The concept of synchronous dataflow scenarios [18] extends the expressive power of SDF by combining streaming data and finite control into a MoC called FSM-SADF [19]. More precisely, application behaviors are clustered into a group of static modes of operation called *scenarios* each modeled by an SDFG, while scenario occurrence patterns are constrained by a nondeterministic FSM. Consider the example FSM-SADF graph (FSM-SADFG) of Fig. 4a. The graph has two scenarios: s_1 and s_2 modeled by two SDFGs. The scenario FSM has two states where each of the states corresponds to one scenario. In the figure, state ϕ_1 corresponds to s_1, while ϕ_2 corresponds to s_2. The scenario FSM defines admissible scenario sequences that can be denoted via the regular expression $(s_1 \mid s_1 s_2)^*$ where $\mid$ denotes a nondeterministic choice. The operational semantics of the model is as follows: every transition in the scenario FSM schedules the execution of one iteration of the SDFG that models the scenarios corresponding to the transition's destination state. We formally define the scenario FSM in Definition 2.

Definition 2 (Scenario FSM)**.** *Given a set S of scenarios, a scenario FSM F on S is a tuple $F = (\Phi, \phi_0, \delta, \Psi)$, where Φ is the set of states, ϕ_0 is the initial state, $\delta \subseteq \Phi \times \Phi$ is the transition relation and $\Psi : \Phi \to S$ is the scenario labeling.*

The formal specification of FSM-SADF follows in Definition 3.

Definition 3 (FSM-SADF)**.** *FSM-SADF* F *is a tuple* $\mathsf{F} = (S, F)$ *where* S *is the set of scenarios and* F *is an FSM on* S*.*

3.5 Max-plus algebra for synchrnous dataflow scenarios

The execution of an FSM-SADFG translates to nothing but a sequence of executions of its constituents, i.e. scenario SDFGs. The sequences are defined by the scenario FSM. A sequence of scenarios can be associated with a sequence of timestamp vectors $\gamma(0), \gamma(1), \ldots$ where

$$\gamma(k+1) = \mathcal{M}(\zeta(k+1)) \otimes \gamma(k). \tag{8}$$

In (8), $\mathcal{M} : S \to \mathbb{R}_{\max}^{|I|\times|I|}$, returns the Max-plus matrix of the scenario SDFG, $\zeta : \mathbb{N}_{>0} \to S$ returns the scenario of the $(k+1)$st FSM-SADFG iteration. Timestamp vector $\gamma(k)$ of the kth FSM-SADFG iteration provides the initial conditions for the execution of the next one. It is known [19], that sequence $\gamma(0), \gamma(1), \ldots$ defined by (8) is an upper bound on the self-timed execution of the respective FSM-SADFG. Fig. 4b shows the execution of the running example FSM-SADFG for the scenario sequence $s_1 s_2 s_1 s_2 s_1 s_1$. One can notice that consecutive scenarios can be active concurrently, i.e. they are pipelined. E.g. at $t = \tau$, three scenarios are active at the same time. An interesting phenomena caused by pipelining that may be puzzling for a noninitiated reader is that token i_5 is recreated in scenario s_2 first, although scenario s_2 always follows s_1. This is due to the fact that in scenario s_2 the firing of A_6 takes place sooner than the corresponding firing of A_6 in scenario s_1. It had been shown in [19] that the completion time of a sequence $\sigma = s_1, \ldots, s_k \in S^*$ of scenarios can be defined as follows

$$\mathcal{A} = \alpha^T \otimes \mu(\sigma) \otimes \beta, \tag{9}$$

where α is the final delay, $\mu : S^* \to \mathbb{R}_{\max}^{n\times n}$ is the morphism that associates sequences of scenarios with Max-plus matrices as follows

$$\mu(\sigma) = \mathcal{M}(s_k) \otimes \ldots \otimes \mathcal{M}(s_1) \tag{10}$$

and β is the initial delay. The initial delay β specifies the initial enabling time of initial tokens and typically $\beta = \mathbf{0}$, while the final delay α serves as a mean to specify the metrics we are interested in. E.g., if we are interested in the makespan of a sequence of scenarios, we set $\alpha = \mathbf{0}$. The triple $\mathcal{A} = (\alpha, \mu, \beta)$ defines a Max-plus automaton [16]. The theory of Max-plus automata had been used in [19] to analyze worst-case performance of FSM-SADFG. We leave this matter aside now and explain it in detail in Section 7. Last thing we bring out is the fact that scenarios do not necessarily need to share the same set of initial tokens which is the case with the running example FSM-SADFG too. In the recursion of (8) and (10), particular scenarios matrices and timestamp vectors need to be expanded so that the right-hand side product remains well-defined. More details can be found in [36]. The intuition behind the formalization of [36] is that every scenario that "misses" some of the initial tokens from the overall union defined over all scenario graphs, needs to be supplemented with the tokens it misses as follows: each missing token is placed on a self-edge of a subsequently added disconnected "dummy" actor. This actor has the firing delay equal to zero.

4 Parameterized dataflow

In this section we elaborate our parameterized dataflow modeling framework which is the crucial milestone towards the definition of parameterized dataflow scenarios that is to follow in Section 5.

Inspired by the approach of [6], we too set up our parametrization framework as a meta-modeling technique that enables integration of parameters and run-time adaptation of parameters into a wide range of dataflow models referred to as base models. The integration of parameters increases the expressive power of the model and renders it capable of capturing fine-grained application dynamics in a succinct manner without departing from the dataflow framework. Within our framework, we

limit our attention to timed uninterpreted base models with conjunctive firing rules [27]. With such models, the actual meaning of computations and semantics of data tokens are not relevant. This means that actor firing rules do not depend on the values of tokens but only on their availability under a conjunctive firing rule. A conjunctive firing rule implies that all actor input channels must contain enough input tokens for the respective actor to fire. In addition, given a particular base model, we require it has a well-defined notion of a graph iteration while parameters are allowed to change only in between iterations. SDF, CSDF and SSDF discussed earlier are all examples of such base models.

Although inspired by it, as we will show, our modeling framework is different than that of [6] as it abstracts from the parameter reconfiguration mechanisms used in that work. These mechanisms are conceived around the *initflow* concept that introduces auxiliary graphs (*init* graph and *subinit* graph) that specify configuration delivery mechanisms to the functional part of the application modeled by the *body* graphs. The addition of these graphs increases complexity and compromises the intuitive appeal of the technique due to complex interactions between the three types of graphs. In our approach we favor a monolithical approach where all actors are treated on equal footing which avoids the need for the partitioning of the structure according to the functionality of particular structure parts. This does not hamper the use of parameter reconfiguration concept as it had been shown that in a parameterized context it can be straightforwardly added to the original structure in a purely dataflow manner [15][3].

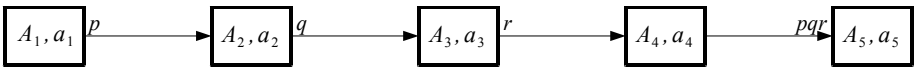
We proceed by defining our parametrization context. By focusing on uninterpreted base models the scope of our parametrization naturally narrows to graph rates and actor firing delays. Both rate parametrization and firing delay parametrization serve the purpose of capturing fine-grained data-dependent application dynamics but at different levels. Rate parametrization concerns data-dependent dynamics that the designer wishes to expose at a module, i.e. actor level. Therefore, parameterized rates capture data-dependent variation of communication patterns between actors. On the other hand, parameterized firing delays are used to capture the temporal effect of data-dependent dynamics within modules represented by actors on the overall composition, i.e. the graph. Moreover, we may say that in this way the designer hides module implementation details irrelevant at a particular abstraction level. E.g., if actors are implemented in software, their firing delays correspond to worst-case execution times (WCET) of modules they represent. These in turn may be represented as parametric expressions including input parameters to the module, or maximal iteration counts of module loops [29].

We parametrize rates using strings containing parameters. We let the string production be governed by an arbitrary grammar $\mathcal{R}$. Similarly, we let actor firing delays be parameterized by an arbitrary grammar $\mathcal{D}$. Using $\mathcal{R}$ and $\mathcal{D}$ we define a parameterized dataflow graph (PDFG) in Definition 4.

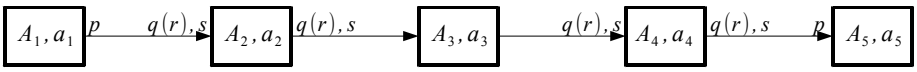
Definition 4 (PDFG)**.** *An PDFG is a tuple* $G = (\mathcal{A}, C, \mathcal{P}_i, \mathcal{P}_d, r, d, i, X_G)$*, where* $\mathcal{A}$ *is the set of actors,* $C \subseteq \mathcal{A} \times \mathcal{A}$ *the set of channels,* $\mathcal{P}_i$ *is the set of nonnegative integer parameters,* $\mathcal{P}_d$ *is the set of nonnegative real parameters,* $r : \mathcal{A} \times C \to \mathcal{R}$ *returns for each port its (possibly symbolic) rate,* $d : \mathcal{A} \to \mathcal{D}$ *returns for each actor its associated (possibly symbolic) firing delay,* $i : C \to \mathbb{N}_0$ *returns for each channel its associated number of initial tokens while* X_G *is the domain of the graph.*

Aside the typical dataflow graph constituents such as actors, channels, rates, firing delays and initial tokens, Definition 4 introduces the concept of PDFG domain adopted from [6]. The domain X_G of an PDFG G is the set of all complete configurations of G. A complete configuration of a PDFG is determined by assigning concrete values to all parameters defined by the sets $\mathcal{P}_i$ and $\mathcal{P}_d$. We denote a configuration of G with $x_i^G \in X_G$ where $i \in \{1, \ldots, |X_G|\}$. Once a configuration is routed through the grammars $\mathcal{R}$ and $\mathcal{D}$ and applied to the PDFG, an instance of that graph emerges, denoted $\iota_G(x_i^G)$. An instance of a PDFG is nothing but a base model graph. E.g., if the parametrization of Definition 4 is applied to SDF, an instance $\iota_G(x_i^G)$ is an SDFG.

Parameterized dataflow paradigm of Definition 4 offers a high modeling flexibility. Firstly, this is due to the fact that rates and firing delays can be expressed via arbitrary expressions containing parameters. Secondly, this is due to the fact that the PDFG domain X_G can be used to define the admissible configurations in a variety of ways, e.g. as a set of mathematical expressions capturing



(a) SPDF specification of an artificial application.



(b) PCSDF specification of an artificial application.

```
while(1){
    fire A1;
    repeat (p) times {
        fire A2;
        repeat (q) times {
            fire A3;
            repeat (r) times {
                fire A4;
            }
        }
    }
    fire A5;
}
```

(c) Schedule for the SPDF specification.

```
while(1){
    fire A1;
    fire A2;
    repeat (q) times {
        repeat (q) times {
            fire A3;
        }
    }
    fire A2;
    repeat (s) times {
        fire A3;
    }
    fire A4;
    fire A5;
}
```

(d) Schedule for the PCSDF specification with $p = qr + s$.

Figure 5: SPDF and PCSDF specifications.

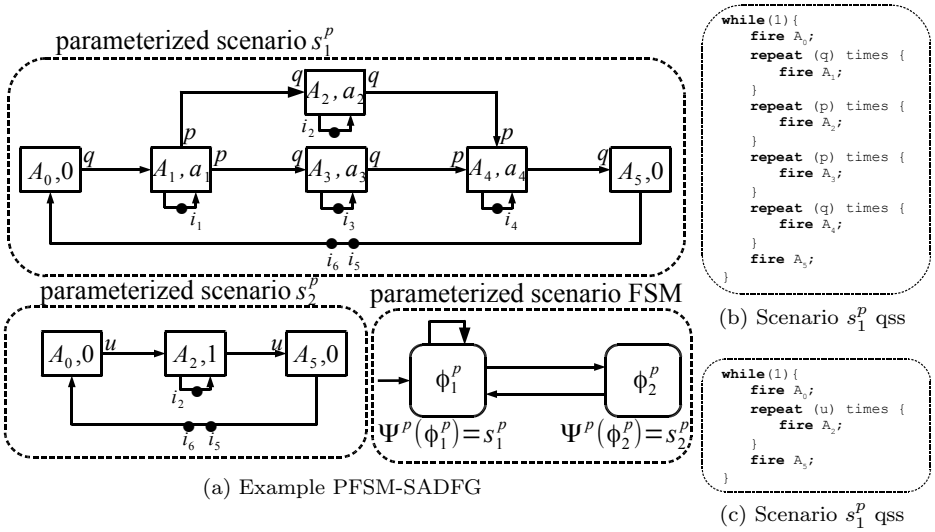
possible complex relationships between graph parameters and characteristics of the input signal and/or the implementation platform.

However, increased flexibility almost immediately implies decreased analyzability. Different works on parameterized dataflow modeling outlined in Section 2, place different restrictions to Definition 4 to make the underlying model fully (e.g. SPDF and BPDF) or at least partly decidable (PSDF). In general, no general decidability criteria can be derived for the structure of Definition 4. Instead, analysis needs to be performed for a particular definition of $\mathcal{R}$.

We exemplify how the parameterized dataflow modeling of Definition 4 can be used to express different existing parameterized dataflow models by merely defining $\mathcal{R}$ in an appropriate way. We use two examples: SPDF introduced in [15] and parameterized CSDF (PCSDF) introduced in [6].

In SPDF, by definition, $\mathcal{R} := k \mid p \mid \mathcal{R}_1 \cdot \mathcal{R}_2$ where $k \in \mathbb{N}_{>0}$ and $p \in \mathcal{P}_i$ with $\mathcal{P}_i$ a set of symbolic variables. Consequently, in SPDF rates are products of positive integers and/or symbolic variables by default constrained to $\mathbb{N}_{>0}$. Fig. 5a shows an example SPDF graph with an arbitrarily defined domain $X_G = \{p \in [1, 100], q \in [1, 12], a_1 = a_2 = a_3 = a_4 = 1.0, a_5 = 2.3x_1 + 3.1x_2, x_1 \in [0.2, 1.9], x_2 = 1.0\}$. An example of a configuration that can be extracted from X_G is $x^G = \{p = 1, q = 3, a_1 = a_2 = a_3 = a_4 = 1, x_1 = 1.0, x_2 = 1.0\}$. Note that X_G enables accommodation of parameters not present in the graph itself but that parameters present in the graph may depend on in various ways. These dependencies are naturally specified by arithmetic expressions, we choose to call constraints. Once x^G is applied, a pure SDFG emerges.

With the given definition of $\mathcal{R}$, existence of a repetition vector that implies consistency for SPDF specifications can be easily verified by a straightforward generalization of the SDF algorithms of [2]. For the example graph, $\Gamma(A_1, A_2, A_3, A_4, A_5) = (1, p, pq, pqr, 1)$. Except for consistency, paper [15] defines criteria for assuring deadlock-freedom and presents an algorithm for schedule generation for SPDF specifications. The schedule for the example graph is displayed in Fig. 5c. Last but not least, SPDF also defines criteria that constraints parameter change patterns so that all rates will always remain well-defined. Within the scope of our work, it is sufficient to say that it is always sate to change parameters in between graph iterations. For more details, we refer the interested reader to [15].



(a) Example PFSM-SADFG

(b) Scenario s_1^p qss

(c) Scenario s_1^p qss

Figure 6: PFSM-SADF.

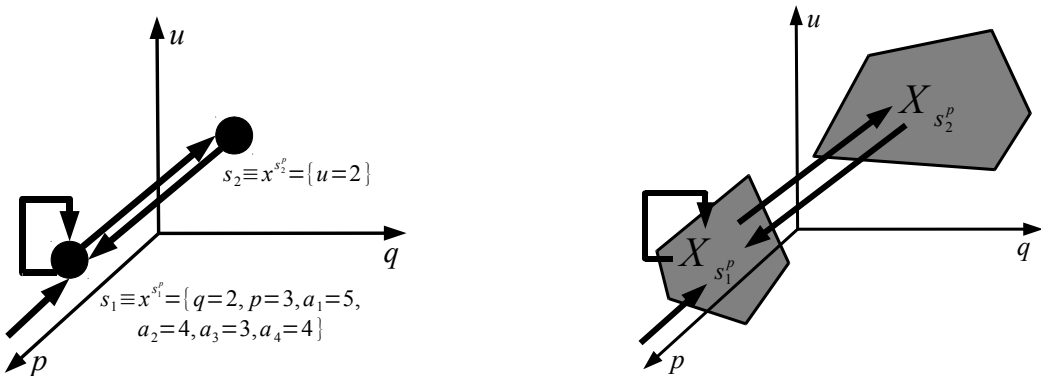
For PCSDF of [6], we set $\mathcal{R}$ as follows $\mathcal{R} := k \mid p \mid \mathcal{R}_1(\mathcal{R}_2) \mid \mathcal{R}_1, \mathcal{R}_2$ where $k \in \mathbb{N}_{>0}$ and $p \in \mathcal{P}_i$ with $\mathcal{P}_i$ a set of symbolic variables by default constrained to $\mathbb{N}_{>0}$. E.g., notation $q(r), s$ for the PCSDF model of Fig. 5b denotes the parameterized cyclo-static dataflow sequence $r, r, r, \ldots, r, s$. In the sequence, r is repeated q times. Notice that the with such a definition of $\mathcal{R}$, we both parametrize the phase (length) and particular phase rates. For the example graph, $X_G = \{p \in [50, 100], r \in [3, 20], q \in \mathbb{N}_{>0}, s \in \mathbb{N}_{>0}, p = qr + s, a_1 = a_2 = a_3 = a_4 = a_5 = 1.0\}$. How to assure boundedness, liveness and how to schedule PCSDF specification is to the best of our knowledge an open problem. For the simple structure of Fig. 5b with a requirement $p = qr + s$ stemming from the graph domain definition, the quasi-static schedule of Fig. 5d can be easily derived.

5 Integration of parameterized dataflow and finite-state machines

In this section we investigate the integration of parameterized dataflow and FSMs in a concept we refer to as parameterized dataflow scenarios. More precisely, we propose a hybrid framework that combines finite-state control expressed via an FSM and streaming data expressed via parameterized dataflow introduced in Section 4. We achieve this by generalizing the concept of FSM-SADF (HDF) to one which allows to express fine-grained data-dependent dynamics and control requirements of modern streaming applications. More precisely, we model the execution of an application as a sequence of parameterized scenarios or shortly scenarios. The sequencing of scenarios is dictated by the control structure captured by the scenario FSM while internal fine-grained data-dependent dynamics of a particular scenario is captured by the scenario PDFG. From now on, we use the terms scenario, PDFG and scenario PDFG interchangeably. We refer to the new model as FSM-based parameterized scenario-aware dataflow (PFSM-SADF).

We exemplify using the structure of Fig. 6a. The composite PFSM-SADF graph (PFSM-SADFG) in the figure is defined over two parameterized scenarios s_1^p and s_2^p, each of which is modeled by a scenario PDFG. Sequencing of scenarios is dictated by the parameterized scenario FSM or shortly scenario FSM, where each state corresponds to one scenario. The example scenario FSM has two states: ϕ_1^p and ϕ_1^p. State ϕ_1^p corresponds to s_1^p and state ϕ_2^p corresponds to s_2^p.

The operational semantics of the PFSM-SADF model is as follows. Each reaction/transition of the scenario FSM incurs the execution of one iteration of an arbitrary instance of the scenario PDFG that the transition destination state corresponds to. An execution of one iteration of the scenario s_i^p PDFG, translates to the execution of one iteration of an arbitrary instance of the scenario s_i^p PDFG



(a) Operational semantics of FSM-SADF.

(b) Operational semantics of PFSM-SADF.

Figure 7: Comparison of operational semantics of FSM-SADF and PFSM-SADF.

defined by some configuration $x^{s_i^p}$ originating from the scenario PDFG domain $X_{s_i^p}$ and denoted $\iota_{s_i^p}(x^{s_i^p})$. Therefore, the model fosters nondeterminism at two levels. The inter-scenario level, where the parameterized scenario to be activated next is chosen and at the intra-scenario level where one of the instances of that scenario is chosen to carry out the actual execution of the scenario. Fig. 7b illustrates the operational semantics of the PFSM-SADF of Fig. 6a. Scenario PDFG domains are depicted as 2-dimensional planes in the $p-q-u$ space (we omit actor firing delay parameters). E.g., every time a transition $\phi_1^p \rightarrow \phi_2^p$ is taken, one iteration of the scenario s_2^p PDFG is executed. This corresponds to the execution of one iteration of an arbitrary instance of scenario s_2^p PDFG defined by the configurations found in the $X_{s_2^p}$ hyperplane.

We use the opportunity to compare the operational semantics of PFSM-SADF to that of FSM-SADF. Consider the FSM-SADFG of Fig. 4a. Note that the FSM-SADFG of Fig. 4a can be obtained by applying configuration $x^{s_1^p} = \{q = 2, p = 3, a_1 = 5, a_2 = 4, a_3 = 3, a_4 = 4\}$ to s_1^p PDFG and by applying the configuration $x = \{u = 2\}$ to s_2^p PDFG of the PFSM-SADFG of Fig. 6a. Thus, the illustration of Fig. 7a that illustrates the operational semantics of the FSM-SADFG of Fig. 4 can be obtained by collapsing the hyperplanes of Fig. 7b to one point. E.g., the transition $\phi_1 \rightarrow \phi_2$ is refined by the execution of one iteration of the scenario s_2 SDFG (which can be obtained by applying the configuration $\{u = 2\}$ to the s_2^p PDFG of the PFSM-SADFG of Fig. 6a). This informally proves that PFSM-SADF generalizes FSM-SADF (HDF) as FSM-SADF can be instantiated from PFSM-SADF if the base model of the scenarios PDFGs is SDF. In that case, a PFSM-SADFG with only one configuration per scenario is nothing but an FSM-SADFG. Moreover, in contrast to FSM-SADF whose underlying concurrency model (SDF) is fully static, PFSM-SADF via parameterized dataflow modeling of Section 4 allows scenario representations that are necessarily not fully static through the use of base models different than SDF. A good example is PCSDF, that when used as the base model of PFSM-SADF is at run-time instantiated into a CSDF graph. A reader may be sceptical towards the claim that the concept of parameterized dataflow scenarios also generalizes HDF because HDF unlike FSM-SADF (where the FSM is flat and sequential) enforces a hierarchical FSM discipline. This is not an issue because as stated in [22], hierarchy adds nothing to the model of computation but is used to reduce the number of transitions and makes the FSM more intuitive and easier to understand.

We formally define the new model. First, we define the parameterized scenario FSM in Definition 5.

Definition 5 (Parameterized scenario FSM)**.** *Given a set S^p of parameterized scenarios, a parameterized scenario FSM F^p over S^p is a tuple $F^p = (\Phi^p, \phi_0^p, \delta^p, \Psi^p)$, where Φ^p is the set of states, ϕ_0^p is the initial state, $\delta^p \subseteq \Phi^p \times \Phi^p$ is the transition relation and $\Psi^p : \Phi^p \rightarrow S^p$ is the scenario labeling.*

Thereafter, we expose the definition of PFSM-SADF.

Definition 6 (PFSM-SADF)**.** *PFSM-SADF* F^p *is a tuple* $\mathsf{F}^p = (S^p, F^p)$ *where* S^p *is the set of SDF-PDF scenarios and* F^p *is an FSM on* S^p*.*

We briefly discuss the consistency, deadlock freedom and scheduling properties of PFSM-SADF. A PFSM-SADF progresses in iterations of its PDFG scenario instances like a FSM-SADF model progresses in iterations of its scenario SDFGs. It is natural to define scenarios at dataflow graph iteration granularity because an iteration typically represents a coherent set of computations, e.g. decoding a video frame. Like in FSM-SADF, the synchronization between consecutive scenarios is achieved using the common initial tokens of those scenarios. These tokens have the same designators (names) across scenarios. This is natural, as scenarios are represented by dataflow graphs and in the parameterized dataflow (PDF) domain of Definition 4, initial tokens represent initial conditions for execution of the graph. Typically, but not required [36], common tokens reside on common channels (a channel that shares the same source and destination actor in both scenarios) and common channels have the same number of initial tokens in all scenarios.

Given the operational semantics of PFSM-SADF and its synchronization principles described above, the same criteria for the consistency of applications modeled by PFSM-SADF as for those modeled by FSM-SADF applies, i.e. consistency of the application is guaranteed if all individual scenarios are consistent [35]. Deadlock freedom is a more subtle concept. Since our modeling targets streaming applications, we consider infinite traces of the scenario FSM. Therefore, an application modeled by PFSM-SADF is deadlock-free if all individual scenarios are deadlock-free and there exists no state in the scenario FSM with no outgoing transitions. As PFSM-SADF is a dynamic dataflow model - it cannot be statically scheduled. Still, as PFSM-SADFG progresses in scenario sequences, given a trace of the scenario FSM a schedule for the trace can be constructed by concatenating schedules of particular scenario graphs. Therefore, we deem a PFSM-SADFG schedulable if for all its scenario graphs considered in isolation a (quasi)-static schedule can be found.

Determining whether a particular scenario PDFG is consistent, deadlock-free and schedulable is not an easy task. In general, no general decidability criteria can be derived for the structure of Definition 4. Instead, analysis needs to be performed for a particular definition of $\mathcal{R}$.

6 Analyzing parameterized synchronous dataflow scenarios

In this section we apply the concept of parameterized dataflow scenarios of Definition 6 to SDF which is the most mature and stable dataflow formalism. We refer to this specialization of PFSM-SADF of Definition 6 as SDF-based PFSM-SADF (SDF-PFSM-SADF) for which we develop worst-case performance analysis algorithms. An example of an SDF-PFSM-SADF graph (SDF-PFSM-SADFG) is shown in Fig. 6a. In SDF-PFSM-SADF, scenarios are represented using SDF-based parameterized dataflow (SDF-PDF), i.e. SDF-PDF graphs (SDF-PDFG).

6.1 SDF-based parameterized dataflow

SDF-PDFGs are basic constituents of our parameterized synchronous dataflow scenarios as they capture the data-dependent dynamics of a SDF-PFSM-SADF scenario. An SDF-PDFG is derived from a general PDFG by applying the definition of $\mathcal{R}$ shown in (11) to Definition 4.

$$\mathcal{R} := k \mid p \mid \mathcal{R}_1 \cdot \mathcal{R}_2. \tag{11}$$

In (11), $k \in \mathbb{N}_{>0}$ and $p \in \mathcal{P}_i$ with $\mathcal{P}_i$ a set of symbolic variables, i.e. rates are defined as products of positive integers and/or symbolic variables by default constrained to $\mathbb{N}_{>0}$. As firing delays do not influence the dataflow behavior of the graph, in principle in Definition 4, we could allow for an arbitrary $\mathcal{D}$, but for simplicity and some purely technical constraints to be explained in Section 7 we choose to consider actor firing delays as linear expressions of parameters as defined by $\mathcal{D}$ of (12).

$$\mathcal{D} := k \mid k \cdot d \mid \mathcal{D}_1 + \mathcal{D}_2. \tag{12}$$

In (12), $k \in \mathbb{R}_{\geq 0}$ and $d \in \mathcal{P}_d$, i.e. actor firing delays can be nonnegative real constants or linear combinations of parameters constrained by default to $\mathbb{R}_{\geq 0}$. With the definition of $\mathcal{R}$ of (11), one can decide on consistency, deadlock-freedom and schedulability of SDF-PDF specifications using the results of [15] and consequently on the same properties of the enveloping SDF-PFSM-SADFG.

6.2 Modeling SDF-PDF with Max-plus

In this section we study how Max-plus algebra can be used to model the temporal behavior of SDF-PDF under self-timed execution.

6.2.1 Opening remarks

In accordance with the Max-plus semantics of SDF [18], as the base model of SDF-PDF, the evolution of an SDF-PDFG G can be given as a recursive Max-plus linear equation relating the timestamps vectors $\gamma(k+1)$ and $\gamma(k)$ of initial tokens after the $(k+1)$st and the kth SDF-PDFG iteration, respectively, as follows

$$\gamma(k+1) = \mathcal{M}_G(\zeta(k+1)) \otimes \gamma(k). \tag{13}$$

In (13), $\mathcal{M}_G : X_G \to \mathbb{R}_{\max}^{|I|\times|I|}$ denotes a mapping that for each $x^G \in X_G$ returns the associated Max-plus matrix of the instance SDFG, i.e. $M_{\iota_G(x^G)}$. Mapping $\zeta : \mathbb{N}_{>0} \to X_G$ returns the configuration that determines the instance $\iota_G(x^G)$ that is executed as the $(k+1)$st iteration of the SDF-PDFG. Therefore, the temporal behavior of an SDF-PDFG, can be fully described by a set of Max-plus matrices of all its instances. The number of such matrices equals to the cardinality of X_G, i.e. $|X_G|$. However, $|X_G|$ is typically very large and proportional to the cardinality of the product set of parameter ranges. This renders the generation of this set via enumeration of X_G often infeasible in practice. This is because of the high time penalty incurred by the generation of these matrices for SDFGs with very large repetition vectors (cf. Fig. 11 in [19]).

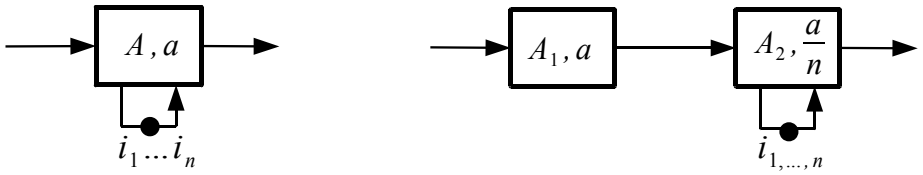
Instead of enumeration, with the overall goal of compacting the representation while retaining relevant information, we advocate for the characterization of temporal behavior of SDF-PDF models using a set of parameterized Max-plus matrices, i.e. matrices whose entries will be parameterized expressions in $\mathcal{P}_i$ and $\mathcal{P}_d$ (cf. Definition 4). In the light of aforementioned, the evolution of an SDF-PDFG can be described via

$$\gamma(k+1) = \underbrace{\left[\mathcal{M}_G^{\text{par}}(\zeta(k+1))\right](\zeta(k+1))}_{\mathcal{M}_G(\zeta(k+1))} \otimes \gamma(k). \tag{14}$$

In (14), $\mathcal{M}_G^{\text{par}} : X_G \to E^{I\times I}$ denotes a mapping that for each $x^G \in X_G$ returns the associated parameterized Max-plus matrix corresponding to a particular $x^G \in X_G$, that when evaluated for that x^G is nothing but the Max-plus matrix of the instance SDFG, i.e. $M_{\iota_G(x^G)}$. Notation E defines the set of all arithmetic expressions defined on $(\mathcal{P}_i \cup \mathcal{P}_d \cup -\infty)$ which in turn is used to define $E^{I\times I}$ the set of all I by I Max-plus matrices with entries in E. The semantics of mapping $\zeta : \mathbb{N}_{>0} \to X$ stays the same as in (13). Once such a parameterized characteristic matrix is evaluated for a concrete configuration that defines the $(k+1)$st iteration of the structure, i.e. at $\zeta(k+1)$, a concrete Max-plus matrix $\mathcal{M}_G(\zeta(k+1))$ emerges, i.e. (14) reduces to (13). Therefore, (14) is a compact representation of (13) developed around the mapping $\mathcal{M}_G^{\text{par}}$. This way, one needs not to perform an enumeration of X_G. The difficulty is moved, however, to determining the mapping $\mathcal{M}_G^{\text{par}}$ defining the collection of parameterized matrices as constituents of its codomain. It is a collection (and not a single parameterized matrix) because in a parametric (general) setting, the partitioning of X_G occurs naturally due to the max operator in Max-plus.

Because SDF is the base model of SDF-PDF, the timestamp t of any token produced within the $(k+1)$st SDF-PDFG iteration can be written as a Max-plus scalar product

$$t = \bigoplus_{i_j} m_j^{\text{par}} \otimes t_{i_j} = [m_1^{\text{par}}, \ldots, m_{|I|}^{\text{par}}] \otimes \gamma(k), \tag{15}$$



(a) Actor with auto-concurrency bounded to n.

(b) Latency-rate abstraction of the actor on the left.

Figure 8: Latency-rate abstraction

where t_{i_j} are the timestamps of initial tokens after the kth graph iteration and m_j^{par} are now parameterized expressions in E. Therefore, the timestamps of initial tokens at the end of the $(k+1)$st iteration embedded in $\gamma(k+1)$, can be denoted as follows

$$t'_{i_l} = \bigoplus_{i_j \in I} m_{l,j}^{\text{par}} \otimes t_{i_j} = [m_{l,1}^{\text{par}}, \ldots, m_{l,|I|}^{\text{par}}] \otimes \gamma(k). \tag{16}$$

In this case, dependency vectors $[m_{l,1}^{\text{par}}, \ldots, m_{l,|I|}^{\text{par}}]$ where $l = 1, \ldots, |I|$ will form the rows of the parameterized Max-plus SDF-PDFG matrix as an element of the codomain of $\mathcal{M}_G^{\text{par}}$. Thus, the challenge lies in determining expressions of type (16). In the remainder of this section we show how to do this for a type of graphs that in addition to being consistent, deadlock free and quasi-static schedulable satisfy the following two requirements.

Requirement 1. *For all SDF-PDFG channels $c \in C$ such that $src(c) \neq dst(c)$ and $i(c) > 0$, $i(c) > \Gamma(dst(c))$ must hold, i.e. if c has initial tokens, there must be enough of them for actor $dst(c)$ to complete all its firings within the iteration. Functions $src : C \to \mathcal{A}$ and $dst : C \to \mathcal{A}$ return for each channel its source and destination actor, respectively.*

With this requirement, we limit our attention to feed-forward structures where initial tokens in graph channels (other than self-edges) are not reproduced more than once within an iteration. This way, in cyclic graphs, feedback loops can be broken resulting in acyclic specifications. Fortunately a large number of streaming applications fall under this requirement that is typically enforced in literature to enable effective quasi-static scheduling [5][15][6]. In the context of our Max-plus analysis we impose this requirement as it is not clear how to deal with schedule loops of length grater than one [2] with parametric repetition counts.

Requirement 2. *For all SDF-PDFG channels $c \in C$ such that $src(c) = dst(c)$, $i(c) = 1$ must hold.*

This requirement disables the bounding of auto-concurrency. Auto-concurrency of actors can be bounded by inserting a particular number of tokens on their self-edges. With Requirement 2 we allow either full auto-concurrency for an actor or no auto-concurrency at all. This is because we during the process of determining $\mathcal{M}_G^{\text{par}}$ with regard to (15) wish to avoid situations where tokens produced by the actor depend on different self-edge tokens from one actor firing to the next. This requirement is not restrictive in practice as any such actor in the graph can be replaced by its latency-rate abstraction [40] that conservatively captures its temporal behavior. Fig. 8b shows such a conservative latency-rate based abstraction of an actor with auto-concurrency bounded to n displayed in Fig. 8a. Note that the collection $i_1, \ldots, i_n$ of Fig. 8a is collapsed into a single token $i_{1,\ldots,n}$ of Fig. 8b. Actor A itself is expanded into two actors A_1 and A_2 with firing delays a and $\frac{a}{n}$, respectively. We believe the same principle could be straightforwardly applied to cyclic graph substructures with channels not compliant to Requirement 1 using the notion of local iterations [15]. The "problematic" subgraphs would then be replaced by their latency-rate abstractions. The procedure could be recursively repeated in a bottom-up fashion in line with different levels of substructure nesting. This is, however, a subject of future work.

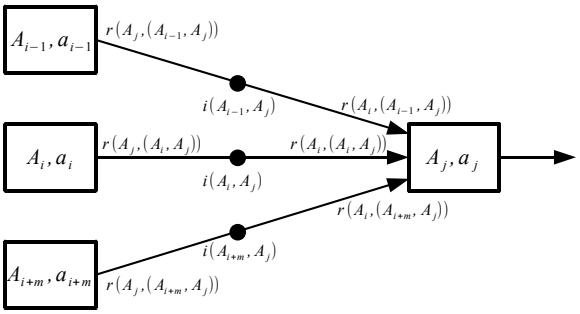
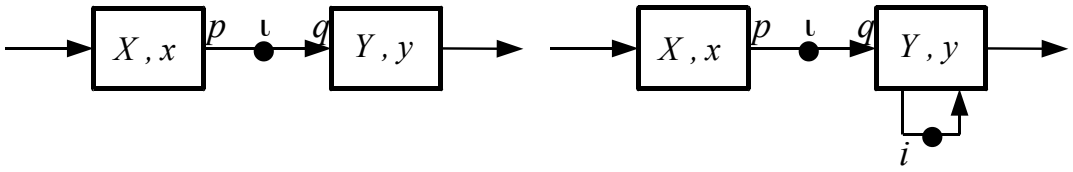


Figure 9: Illustration of (17).



(a) Without self-edges.

(b) With self-edges.

Figure 10: Relevant SDF-PDF structures.

6.2.2 Max-plus model of SDF-PDF execution

To determine $\mathcal{M}_G^{\text{par}}$, as with SDF [18], we need to compute one iteration of the considered SDF-PDFG, i.e. the production times of restored initial tokens after one iteration of the graph expressed via the scalar product of (16).

With SDF it is straightforward to keep track of timestamps of tokens produced by actor firings on channels within a simple FIFO container. This is due the fact the channel quantities are finite and known. Each FIFO element stores the dependency vector of the token it refers to (cf. (5)). With parameterized rates, the situation is more subtle. The channel quantities will still be finite but unknown as they are determined by parameters. Therefore, it would become cumbersome to define such a FIFO structure. Instead, we capture the ordering of tokens using the firing indices of their producing actors as a mapping $\tau : \mathcal{A} \times \mathbb{Z} \to E^{n\times 1}$. The nonpositive firing indices are reserved for the initial tokens themselves. As initial tokens represent the initial conditions for the execution of the graph, they are therefore assumed to be produced by some past actor firing. We give the following definition of τ time as follows from the Max-plus algebraic semantics of self-timed execution of SDF that SDF-PDF inherits:

$$\tau(A_j, n) = \bigoplus_{A_i|(A_i,A_j)\in C} \tau\left(A_i, \left\lceil \frac{n \cdot r(A_j,(A_i,A_j)) - i((A_i,A_j))}{r(A_i,(A_i,A_j))} \right\rceil\right) \otimes d(A_j). \tag{17}$$

The timestamps of tokens produced by the nth firing of actor A_j will correspond to the maximal arrival times of all input tokens (over all input channels) to be consumed by the firing (synchronization) increased by the firing delay of A_j itself (delay). Notice that the firing indices of feeding actors need to be adjusted according to the respective channel rates and any presence of initial tokens. An example SDF-PDF structure illustrating the semantics of (17) is available in Fig. 9.

Equation (17) reveals that in computing the response of an actor, different inputs (channels) can be treated in isolation. Ultimately, particular contributions need to be superposed. This corresponds to the Max-plus superposition principle [1]. We first show how to apply (17) to one input channel. For that purpose, we consider a minimal but a general (for the purpose) SDF-PDF structure of Fig. 10. We compute the response of actor Y for two cases. First (on the left), when Y has no self-edge. Second (on the right), when Y has a self-edge with one initial token. These are the only two types of actors allowed in our input specifications (cf. Requirement 2). Furthermore, we only

treat the case when in the figure, $\iota = 0$. The case where $\iota > 0$ is trivial due to Requirement 1. More precisely, within one iteration of the graph, actor's demand for input tokens on that channel will always refer to one of those ι initial tokens, i.e. no firings of X within the iteration need to be considered. We apply (17) to the structure of Fig. 10a. We obtain

$$\tau(Y, n) = \tau(X, \lceil \frac{n \cdot q}{p} \rceil) \otimes y. \tag{18}$$

Equation (18) reveals that in the case when Y has no self-edge, Y only delays the input by y. A more interesting case arises when Y has a self-edge with one initial token (cf. Fig 10b). In this case, (17) transforms to

$$\tau(Y, n) = \left(\tau(Y, n-1) \oplus \tau(X, \lceil \frac{n \cdot q}{p} \rceil) \right) \otimes y. \tag{19}$$

We treat (19) using *backward substitution*. Backward substitution is a well-known method for solving recurrence equations and it works exactly as the name implies. In particular, starting from the equation itself, we work backwards substituting the values of the recurrence for previous ones. E.g., if we substitute

$$\tau(Y, n-1) = \left(\tau(Y, n-2) \oplus \tau(X, \lceil \frac{(n-1) \cdot q - \iota}{p} \rceil) \right) \otimes y \tag{20}$$

into (19), we obtain

$$\begin{aligned} \tau(Y, n) &= \left(\left(\tau(Y, n-2) \oplus \tau(X, \lceil \frac{(n-1) \cdot q - \iota}{p} \rceil) \right) \otimes y \oplus \tau(X, \lceil \frac{n \cdot q - \iota}{p} \rceil) \right) \otimes y \\ &= \tau(Y, n-1) \otimes y^{\otimes 2} \oplus \tau(X, \lceil \frac{(n-1) \cdot q - \iota}{p} \rceil) \otimes y^{\otimes 2} \oplus \tau(X, \lceil \frac{n \cdot q - \iota}{p} \rceil) \otimes y. \end{aligned} \tag{21}$$

If we continue, i.e. unfold (19) for k times and substitute it back, we obtain

$$\tau(Y, n) = \tau(Y, n-k) \otimes y^{\otimes n} \oplus \bigoplus_{i=1}^{k} \tau(X, \lceil \frac{(n-i+1) \cdot q - \iota}{p} \rceil) \otimes y^{\otimes i}. \tag{22}$$

We obtain the base case when $k = n$ from (22) as follows

$$\tau(Y, n) = \tau(Y, 0) \otimes y^{\otimes n} \oplus \underbrace{\bigoplus_{i=1}^{n} \tau(X, \lceil \frac{(n-i+1) \cdot q - \iota}{p} \rceil) \otimes y^{\otimes i}}_{conv(\tau(X, \lceil \frac{n \cdot q - \iota}{p} \rceil), y^{\otimes n})}. \tag{23}$$

In the second term of the Max-plus summation of (23) we recognize the Max-plus convolution of the input token timestamp sequence and the impulse response of actor Y, denoted $h(Y, n)$ where $h : \mathbb{N}_{>0} \to \mathbb{R}_{\max}$ is the timestamp sequence belonging to tokens produced by the actor in response to the impulse input token timestamp sequence

$$u(n) = \begin{cases} 0 & \text{if } n = 1 \\ -\infty & \text{otherwise} \end{cases}, \text{for all } n \in \mathbb{N}_{>0}. \tag{24}$$

For a complete presentation we refer to [17]. When the actor has a self-edge with one initial token, its impulse response takes the form

$$h(Y, n) = y^{\otimes n}, \text{for all } n \in \mathbb{N}_{>0}. \tag{25}$$

When an actor is without a self-edge,

$$h(Y, n) = \begin{cases} y & \text{if } n = 1 \\ -\infty & \text{otherwise} \end{cases}, \text{for all } n \in \mathbb{N}_{>0}. \tag{26}$$

An actor without a self-edge can be interpreted as an actor with a self-edge with an infinite stock of initial tokens all available at $t = -\infty$. Then, when (26) is applied to (23), (23) reduces to (18). We formally define the previously used concept of Max-plus convolution.

Definition 7. *Let $\sigma_1(n)$ and $\sigma_2(n)$ be two sequences in $\mathbb{R}_{\max}$, i.e. $\sigma_{1,2} : \mathbb{N}_{>0} \to \mathbb{R}_{\max}$. The convolution of the two, denoted $conv(\sigma_1, \sigma_2)$ is defined as*

$$conv(\sigma_1, \sigma_2)(n) = \bigoplus_{i=1}^{n} \sigma_1(n-i+1) \otimes \sigma_2(i). \tag{27}$$

To tightly bound the Max-plus convolution of (23) using a closed form expression, we use the proposition to follow.

Proposition 1. *Let $\sigma_1(n)$ and $\sigma_2(n)$ be two periodic sequences in $\mathbb{R}_{\max}$ such that $\sigma_1(n) = \delta_1 \otimes {\pi_1}^{\otimes \lceil r \cdot n \rceil}$ and $\sigma_2(n) = {\pi_2}^{\otimes n}$, where $n \in \mathbb{N}_{>0}$, $r \in \mathbb{Q}_{\geq 0}$ and $\delta_1, \pi_1, \pi_2 \in \mathbb{R}_{\max}$. Then, the following inequality holds:*

$$conv(\sigma_1, \sigma_2)(n) < \begin{cases} \delta_1 \otimes {\pi_1}^{\otimes(1+r)} \otimes {\pi_2}^{\otimes n} & \text{if } \pi_2 \geq r \cdot \pi_1 \\ \delta_1 \otimes \pi_2 \otimes {\pi_1}^{\otimes(1+r \cdot n)} & \text{if } \pi_2 \leq r \cdot \pi_1 \end{cases} \tag{28}$$

Proof. We prove this by induction using the argument

$$x \leq \lceil x \rceil < x + 1. \tag{29}$$

First, we consider the case where $\pi_2 \geq r \cdot \pi_1$. We prove the induction base case, i.e. when $n = 1$. By substituting $\sigma_1(n) = \delta_1 \otimes {\pi_1}^{\otimes \lceil r \cdot n \rceil}$ and $\sigma_2(n) = {\pi_2}^{\otimes n}$ into (27), we obtain

$$conv(\sigma_1, \sigma_2)(n) = \delta_1 \otimes \bigoplus_{i=1}^{n} {\pi_1}^{\otimes \lceil r \cdot (n-i+1) \rceil} \otimes {\pi_2}^{\otimes i}. \tag{30}$$

For $n = 1$, (30) reduces to

$$conv(\sigma_1, \sigma_2)(1) = \delta_1 \otimes {\pi_1}^{\otimes \lceil r \rceil} \otimes \pi_2. \tag{31}$$

By combining (29) and (31) we obtain the following inequality

$$conv(\sigma_1, \sigma_2)(1) = \delta_1 \otimes {\pi_1}^{\otimes \lceil r \rceil} \otimes \pi_2 < \delta_1 \otimes {\pi_1}^{\otimes(1+r)} \otimes \pi_2 \tag{32}$$

that proves the case case. We continue with the induction step, i.e. evaluate (30) for $(n+1)$ with the induction hypothesis of (28) where $\pi_2 \geq r \cdot \pi_1$. We obtain

$$\begin{aligned} conv(\sigma_1, \sigma_2)(n+1) &= \delta_1 \otimes \bigoplus_{i=1}^{n+1} {\pi_1}^{\otimes \lceil r \cdot (n-i+2)) \rceil} \otimes {\pi_2}^{\otimes i} \\ &= \delta_1 \otimes \bigoplus_{i=1}^{n} {\pi_1}^{\otimes \lceil r \cdot (n-i+1) \rceil} \otimes {\pi_2}^{\otimes i} \oplus \delta_1 \otimes {\pi_1}^{\otimes \lceil r \rceil} \otimes {\pi_2}^{\otimes(n+1)} \\ &= \delta_1 \otimes conv(\sigma_1, \sigma_2)(n) \oplus \delta_1 \otimes {\pi_1}^{\otimes \lceil r \rceil} \otimes {\pi_2}^{\otimes(n+1)}. \end{aligned} \tag{33}$$

By substituting the induction hypothesis into (33) we obtain the following inequality

$$conv(\sigma_1, \sigma_2)(n+1) \leq \delta_1 \otimes {\pi_1}^{\otimes(1+r)} \otimes {\pi_2}^{\otimes n} \oplus \delta_1 \otimes {\pi_1}^{\otimes \lceil r \rceil} \otimes {\pi_2}^{\otimes(n+1)}. \tag{34}$$

If we use (29) to get rid of the ceiling in (34), we obtain

$$\begin{aligned} &conv(\sigma_1, \sigma_2)(n+1) < \delta_1 \otimes {\pi_1}^{\otimes(1+r)} \otimes {\pi_2}^{\otimes n} \oplus \delta_1 \otimes {\pi_1}^{\otimes(1+r)} \otimes {\pi_2}^{\otimes(n+1)} \\ &conv(\sigma_1, \sigma_2)(n+1) < \delta_1 \otimes {\pi_1}^{\otimes(1+r)} \otimes {\pi_2}^{\otimes(n+1)}. \end{aligned} \tag{35}$$

Inequality (35) shows that the induction hypothesis holds for $(n+1)$ too which completes the proof for the case where $\pi_2 \geq r \cdot \pi_1$. The proof procedure for the remaining case, i.e. $\pi_2 \leq r \cdot \pi_1$ is similar. □

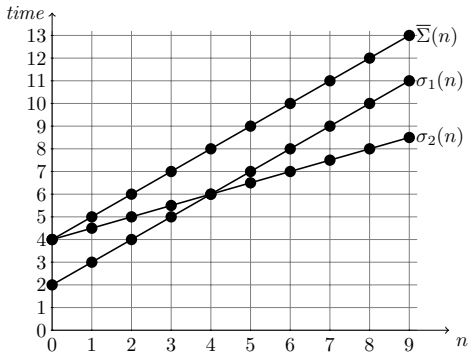


Figure 11: Periodic sequences in $\mathbb{R}_{\max}$.

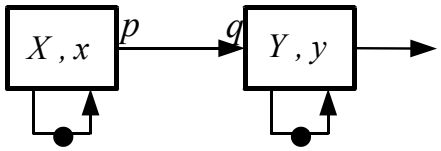


Figure 12: Convolution illustration structure.

Proposition 1 operates on periodic sequences in $\mathbb{R}_{\max}$ and defines a tight bound on the convolution of the two. The bound attains the period (growth rate) of the sequence with the lower period (the slower one). In addition, it is conservatively delayed to account for the noncontinuity of the ceiling function using the relation $x \leq \lceil x \rceil < x + 1$. For a growing value of n the relative error due to conservativity decreases.

A more intuitive interpretation of Proposition 1 can be adopted from [12]. In Max-plus, as in conventional linear system theory the output of a system can be calculated as a convolution between the input to the system and its impulse response. If both input and the impulse response are periodic, then they can be represented in the $\mathbb{N}_{>0} \times \mathbb{R}$ plane (where the x-axis is the index domain and y-axis is the time domain) by a line with any rational slope (r in Proposition 1) shifted along the two axis. E.g., in Fig. 11, the sequence $\sigma_1(n)$ is shifted along the y-axis by 2 and has a slope of 1.

The slope represents the asymptotic growth rate. Their counterpart in conventional linear system theory are sine functions, while their slope is counter-parted with the frequency of those sine waves. If the input slope is strictly smaller than the slope of the impulse response (smaller slope means faster input rate), then the output of the system will be a periodic sequence that inherits the slope of the impulse response and this is a kind of the "lowpass" effect. If opposite, the slope of the output will attain the slope of input.

These are in essence two cases that Proposition 1 treats. We demonstrate the application of Proposition 1 to the structure of Fig. 12. With regard to Proposition 1, in the figure, $\delta_1 = 0, \pi_1 = x, \pi_2 = y, r = \frac{q}{p}$. Fig. 13 shows the execution of the structure with: $p = 3, q = 2, x = 3, y = 5$. In

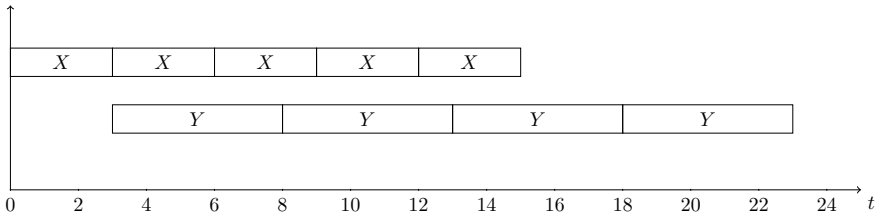


Figure 13: Self-timed execution of the graph of Fig. 12 with $p = 3, q = 2, x = 3, y = 5$.

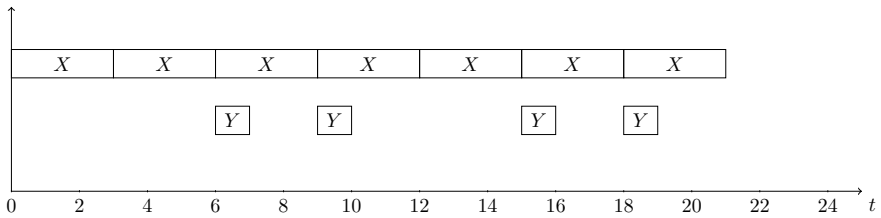


Figure 14: Self-timed execution of the graph of Fig. 12 with $p = 2, q = 3, x = 3, y = 1$.

this case, with regard to Proposition 1, $\pi_2 \geq r \cdot \pi_1$, i.e. actor Y is the bottleneck and its firing delay defines the period of the token sequence produced. According to Proposition 1, this sequence can then be bounded with

$$\tau(Y, n) = 3^{\otimes(1+\frac{2}{3})} \otimes 5^{\otimes n}. \tag{36}$$

On the other hand, consider the execution of the structure when $p = 3, q = 2, x = 3, y = 1$ shown in Fig. 14. In this case, with regard to Proposition 1, $\pi_2 \leq r \cdot \pi_1$, i.e. actor X is the bottleneck and its firing delay normalized to r defines the period of the output token sequence (defined by Y's output channels). According to Proposition 1, this sequence can then be bounded with

$$\tau(Y, n) = 1 \otimes 3^{\otimes(1+\frac{3}{2}\cdot n)}. \tag{37}$$

Equation 37 shows the importance of the added delay (addition of 1 in the Max-plus exponent) to account for the noncontinuity of the ceiling function. Namely, without it, (37) transforms to

$$\tau(Y, n) = 1 \otimes 3^{\otimes(\frac{3}{2}\cdot n)}. \tag{38}$$

which would then underaproximate the sequence of Fig. 14, which is not what we wish in the scope of worst-case temporal behavior analysis.

From Proposition 1 and the fact that the impulse response of an actor with a self-edge with one token is a periodic sequence too (cf. (25)) it follows that the response of a arbitrary actor A_k of an SDF-PDFG compliant to Requirements 1 and 2 within an iteration can be tightly bounded by a delay-period (δ, π) abstraction as follows

$$\begin{aligned} \tau(A_k, n) &= \bigoplus_{i_j} (\delta_{k,j} \otimes {\pi_{k,j}}^{\otimes n}) \otimes t_{i_j} \\ &= [(\delta_{k,1} \otimes {\pi_{k,1}}^{\otimes n}), \ldots, (\delta_{k,|I|} \otimes {\pi_{k,|I|}}^{\otimes n})] \otimes \gamma(k). \end{aligned} \tag{39}$$

A delay-period (δ, π) abstraction defines the dependency vector entries as linear functions of n, i.e. the actor firing index. Given a dependency vector entry that represents the minimal temporal distance of some arbitrary token and an initial token, its period will be determined by the scaled (via rate ratios) firing delay of the slowest actor in the path defined by the producing actors of the two tokens. A delay-period (δ, π) abstraction of (39) defines a tight bound due to Requirement 1 that renders the graph acyclic within an iteration. In case of an actor with multiple input channels, ultimately, the contributions of different input channels need to be superposed. Per dependency vector entry of each contribution, the following propositions defines a tight bound for the output dependency vector entry.

Proposition 2. *Let $S = \{\sigma_1(1), \ldots, \sigma_N(n)\}$ be a set of periodic sequences in $\mathbb{R}_{\max}$ such that $\sigma_i(n) = \delta_i \otimes {\pi_i}^{\otimes n}$. Let $\delta = \max(\delta_1, \ldots, \delta_N)$ and $\pi = \max(\pi_1, \ldots, \pi_N)$ and let $\Sigma(n) = \bigoplus_i \sigma_i(n)$. Then, Σ attains the following periodic conservative tight bound for all $n \in \mathbb{N}_{>0}$*

$$\overline{\Sigma}(n) = \delta \otimes \pi^{\otimes n}. \tag{40}$$

Proof. Follows straightforwardly from the illustration of Fig. 11. By taking π for the growth rate (period) of $\overline{\Sigma}(n)$ and 0 for the initial y-axis shift (delay), $\overline{\Sigma}(n)$ will eventually (for some $n_0 \in \mathbb{N}_{>0}$) compensate for the initial y-axis shift (delay) difference. By taking δ for the delay of $\overline{\Sigma}(n)$, $\overline{\Sigma}(n)$ will dominate for all $n \in \mathbb{N}_{>0}$. □

The result of (40) defines a periodic sequence that attains the period of the slowest input sequence (highest period) and the delay of the most delayed sequence. By using (18) and (23) in conjunction with Proposition 1 and Proposition 2 we can compute the response of any graph actor within an iteration expressed in the form of (39) that we use to render $\mathcal{M}_G^{\text{par}}$. We show how to do this in the section to follow.

6.2.3 Example

We exemplify using the SDF-PDF specification of scenario s_1^p of the PFSM-SADFG of Fig. 6a. We compute the iteration using the schedule of Fig. 6b. The timestamp vector of the kth scenario SDF-PDFG iteration is specified as

$$\gamma(k) = [t_{i_1}, t_{i_2}, t_{i_3}, t_{i_4} t_{i_5}, t_{i_6}]. \tag{41}$$

Furthermore, every entry of $\gamma(k)$ can be written in terms of (17) by associating it with the corresponding producing actor and onwards as the Max-plus scalar product of (15).

$$t_{i_1} = \tau(A_1, 0) = [0, -\infty. - \infty, -\infty, -\infty, -\infty] \otimes \gamma(k), \tag{42}$$
$$t_{i_2} = \tau(A_2, 0) = [-\infty, 0, -\infty, -\infty, -\infty, -\infty] \otimes \gamma(k), \tag{43}$$
$$t_{i_3} = \tau(A_3, 0) = [-\infty, -\infty, 0, -\infty, -\infty, -\infty] \otimes \gamma(k), \tag{44}$$
$$t_{i_4} = \tau(A_4, 0) = [-\infty, -\infty, -\infty, 0, -\infty, -\infty] \otimes \gamma(k), \tag{45}$$
$$t_{i_5} = \tau(A_5, 0) = [-\infty, -\infty, -\infty, -\infty, 0, -\infty] \otimes \gamma(k), \tag{46}$$
$$t_{i_6} = \tau(A_5, -1) = [-\infty, -\infty, -\infty, -\infty, -\infty, 0] \otimes \gamma(k). \tag{47}$$

According to the quasi-static schedule of the specification of Fig. 6b, actor A_0 fires first. It has only one input channel (A_5, A_0), i.e. we need to consider only one contribution. It fires a nonparametric number of times within an iteration and does not have a self-edge. Channel (A_5, A_0) forms a directed cycle (feedback loop) and due to Requirement 1 it must host a sufficient number of initial tokens to fire A_0 at least $\Gamma(A_0)$ times. This is the case because $\Gamma(A_0) = 1$ and $i((A_5, A_0)) = 2$. Thus, we use (18) to compute $\tau(A_0, n)$ for all $n = 1, \ldots, \Gamma(A_0)$ one-by-one. In (18), the right hand side will always refer to one of the timestamps of initial tokens. This way the feedback is effectively broken. We evaluate (18) for A_0 with $n = 1$ using (47).

$$\tau(A_0, 1) = \tau(A_5, -1) \otimes 0 = [-\infty, -\infty, -\infty, -\infty, -\infty, 0] \otimes \gamma(k). \tag{48}$$

A more interesting case is that of actor A_1 which fires a parametric number of times within an iteration, i.e. $\Gamma(A_1) = q$ and has a self-edge. Then $\tau(A_1, n)$ cannot be derived in a one-by-one fashion, but we need to express the analytical relation between $\tau(A_1, n)$ and n. We use (23) in the context of A_1. The input token sequence to A_1 is defined by the vector (48). By substituting (48) and (42) into (23) we obtain

$$\begin{aligned}\tau(A_1, n) &= [0, -\infty, -\infty, -\infty, -\infty, -\infty] \otimes \gamma(k) \otimes {a_1}^{\otimes n} \\ &\oplus [-\infty, -\infty, -\infty, -\infty, -\infty, 0] \otimes \gamma(k) \otimes {a_1}^{\otimes n} \\ &= [{a_1}^{\otimes n}, -\infty, -\infty, -\infty, -\infty, {a_1}^{\otimes n}] \otimes \gamma(k).\end{aligned} \tag{49}$$

Even a more intriguing case arises in the consideration of the next actor in the quasi-static schedule, i.e. A_2 with $\Gamma(A_2) = p$. Actor A_2 has a self-edge and one input dependency defined by tokens produced by A_1 on channel (A_1, A_2). From (23) using (49) we derive

$$\begin{aligned}\tau(A_2, n) = &[-\infty, 0, -\infty, -\infty, -\infty, -\infty] \otimes \gamma(k) \otimes {a_2}^{\otimes n} \\ &\oplus [conv({a_1}^{\otimes \lceil \frac{q}{p} \cdot n \rceil}, {a_2}^{\otimes n}), -\infty, -\infty, -\infty, -\infty, conv({a_1}^{\otimes \lceil \frac{q}{p} \cdot n \rceil}, {a_2}^{\otimes n})] \otimes \gamma(k).\end{aligned} \tag{50}$$

Every entry of the dependency vector of (50) needs to be treated by Proposition 1 one-by-one. For an entry, Proposition 1 gives rise to two cases that split the original parameter space (the graph

domain) into two exclusive parts. For each of these parts, the iteration computation continues in a separate branch. In case of actor A_2 only one split will occur as for the two dependency vector entries of (50) defined by a convolution, these convolutions are identical. The parts of the parameters space to be considered are defined by inequalities $p \cdot a_2 \geq q \cdot a_1$ and $p \cdot a_2 \leq q \cdot a_1$. We continue the computation in the part of the space defined with

$$C_1 \equiv p \cdot a_2 \geq q \cdot a_1. \tag{51}$$

With (51), using (28), (50) transforms to

$$\tau(A_2, n) = [{a_1}^{\otimes(\frac{q}{p}+1)} \otimes {a_2}^{\otimes n}, {a_2}^{\otimes n}, -\infty, -\infty, -\infty, {a_1}^{\otimes(\frac{q}{p}+1)} \otimes {a_2}^{\otimes n}] \otimes \gamma(k). \tag{52}$$

A similar case to that of actor A_2 is the one of actor A_3 with $\Gamma(A_3) = p$. From (23) using (49) we derive

$$\begin{aligned}\tau(A_3, n) = {} & [-\infty, -\infty, 0, -\infty, -\infty, -\infty] \otimes \gamma(k) \otimes {a_3}^{\otimes n} \\ & \oplus [conv({a_1}^{\otimes\lceil\frac{q}{p}\cdot n\rceil}, {a_3}^{\otimes n}), -\infty, -\infty, -\infty, -\infty, conv({a_1}^{\otimes\lceil\frac{q}{p}\cdot n\rceil}, {a_3}^{\otimes n})] \otimes \gamma(k).\end{aligned} \tag{53}$$

Equation (53) mandates a further split of the parameter space with $p \cdot a_3 \geq q \cdot a_1$ and $p \cdot a_3 \leq q \cdot a_1$. We proceed with

$$C_2 \equiv p \cdot a_3 \leq q \cdot a_1. \tag{54}$$

With (54),using (28), (53) reduces to

$$\tau(A_3, n) = [a_1 \otimes a_3 \otimes {a_1}^{\otimes(\frac{q}{p}\cdot n)}, -\infty, {a_3}^{\otimes n}, -\infty, -\infty, a_1 \otimes a_3 \otimes {a_1}^{\otimes(\frac{q}{p}\cdot n)}] \otimes \gamma(k). \tag{55}$$

We proceed with actor A_4 with $\Gamma(A_4) = q$. This actor has two input channels, and therefore to compute its outputs we apply the Max-plus superposition principle. First we consider the contribution of channel (A_2, A_4) whose token timestamps are defined by (52). By substituting (52) into (23) we obtain

$$\begin{aligned}\tau^{(A_2,A_4)}(A_4, n) = {} & [-\infty, -\infty, -\infty, 0, -\infty, -\infty] \otimes \gamma(k) \otimes {a_4}^{\otimes n} \\ & \oplus [{a_1}^{\otimes(1+\frac{q}{p})} \otimes conv({a_2}^{\otimes\lceil\frac{p}{q}\cdot n\rceil}, {a_4}^{\otimes n}), conv({a_2}^{\otimes\lceil\frac{p}{q}\cdot n\rceil}, {a_4}^{\otimes n}), -\infty, \\ & -\infty, -\infty, {a_1}^{\otimes(1+\frac{q}{p})} \otimes conv({a_2}^{\otimes\lceil\frac{p}{q}\cdot n\rceil}, {a_4}^{\otimes n})] \otimes \gamma(k).\end{aligned} \tag{56}$$

The convolutions of (56) split the parameter space with $q \cdot a_4 \geq p \cdot a_2$ and $q \cdot a_4 \leq p \cdot a_2$. We arbitrarily choose to proceed with

$$C_3 \equiv q \cdot a_4 \leq p \cdot a_2. \tag{57}$$

With (57), via (28), (56) becomes

$$\begin{aligned}\tau^{(A_2,A_4)}(A_4, n) = {} & [{a_1}^{\otimes(1+\frac{q}{p})} \otimes a_2 \otimes a_4 \otimes {a_2}^{\otimes(\frac{p}{q}\cdot n)}, a_2 \otimes a_4 \otimes {a_2}^{\otimes(\frac{p}{q}\cdot n)}, \\ & -\infty, {a_4}^{\otimes n}, -\infty, {a_1}^{\otimes(1+\frac{q}{p})} \otimes a_2 \otimes a_4 \otimes {a_2}^{\otimes(\frac{p}{q}\cdot n)}] \otimes \gamma(k).\end{aligned} \tag{58}$$

Similarly, we calculate the contribution of channel (A_3, A_4).

$$\begin{aligned}\tau^{(A_3,A_4)}(A_4, n) = {} & [-\infty, -\infty, -\infty, 0, -\infty, -\infty] \otimes \gamma(k) \otimes {a_4}^{\otimes n} \\ & \oplus [conv(a_1 \otimes a_3 \otimes {a_1}^{\otimes(\frac{q}{p}\cdot\lceil\frac{p}{q}\cdot n\rceil)}, {a_4}^{\otimes n}), -\infty, conv({a_3}^{\otimes(\lceil\frac{p}{q}\cdot n\rceil)}, {a_4}^{\otimes n}), \\ & -\infty, -\infty, conv(a_1 \otimes a_3 \otimes {a_1}^{\otimes(\frac{q}{p}\cdot\lceil\frac{p}{q}\cdot n\rceil)}, {a_4}^{\otimes n})] \otimes \gamma(k)\end{aligned} \tag{59}$$

In (59), two convolutions define the further split in the parameter space. The first one (first and last entries of the dependency vector) proposes the following split: $a_4 \geq a_1$ and $a_4 \leq a_1$. We proceed with the option

$$C_4 \equiv a_4 \leq a_1. \tag{60}$$

With (60), (59) becomes

$$\tau^{(A_3,A_4)}(A_4,n) = [{a_1}^{\otimes(1+\frac{q}{p})} \otimes a_3 \otimes a_4 \otimes {a_1}^{\otimes n}, -\infty, conv({a_3}^{\otimes(\lceil \frac{p}{q}\cdot n\rceil)}, {a_4}^{\otimes n}), {a_4}^{\otimes n}, \\ -\infty, {a_1}^{\otimes(1+\frac{q}{p})} \otimes a_3 \otimes a_4 \otimes {a_1}^{\otimes n}] \otimes \gamma(k). \tag{61}$$

The remaining convolution of (61) defines a further split along the current branch of exploration via options $q \cdot a_4 \geq p \cdot a_3$ and $q \cdot a_4 \leq p \cdot a_3$. By selecting

$$C_5 \equiv q \cdot a_4 \leq p \cdot a_3, \tag{62}$$

from (61) we derive

$$\tau^{(A_3,A_4)}(A_4,n) = [{a_1}^{\otimes(1+\frac{q}{p})} \otimes a_3 \otimes a_4 \otimes {a_1}^{\otimes n}, -\infty, a_3 \otimes a_4 \otimes {a_3}^{\otimes(\frac{p}{q}\cdot n)}, {a_4}^{\otimes n}, \\ -\infty, {a_1}^{\otimes(1+\frac{q}{p})} \otimes a_3 \otimes a_4 \otimes {a_1}^{\otimes n}] \otimes \gamma(k). \tag{63}$$

To finalize the computation of response of A_4 now we need to superpose the contributions of A_2 and A_4. Thus

$$\tau(A_4,n) = \tau^{(A_2,A_4)}(A_4,n) \oplus \tau^{(A_3,A_4)}(A_4,n). \tag{64}$$

The computation of (64) involves an iterative approach where each entry of the dependency vector of the first contribution is paired with the corresponding entry of the second contribution and the combination is treated by Proposition 2. By considering all combinations of maximal delays and periods between corresponding dependency vector entries, Proposition 2 further splits the parameter space. For the concrete examples of (58) and (63) only the first and the sixth entry of dependency vectors are to be treated as the others are either equal (the fourth and the fifth entry) or one immediately dominates because the other equals to $-\infty$ (second and third entry). For the first and the sixth entry (note these are equal in the respective dependency vectors of (58) and (63)) in the light of Proposition (2) we define the corresponding delay-period abstractions as follows

$$\delta_1 = {a_1}^{\otimes(1+\frac{q}{p})} \otimes a_2 \otimes a_4 \qquad \pi_1 = {a_2}^{\otimes(\frac{p}{q})} \tag{65}$$

$$\delta_2 = {a_1}^{\otimes(1+\frac{q}{p})} \otimes a_3 \otimes a_4 \qquad \pi_2 = a_1 \tag{66}$$

Now, Proposition 2 mandates four splits in the parameter space defined as follows

$$\left.\begin{array}{l}\delta = \delta_1 = \max(\delta_1,\delta_2),\\ \pi = \pi_1 = \max(\pi_1,\pi_2)\end{array}\right\} \ \delta_1 \geq \delta_2, \pi_1 \geq \pi_2 \tag{67}$$

$$\left.\begin{array}{l}\delta = \delta_1 = \max(\delta_1,\delta_2),\\ \pi = \pi_2 = \max(\pi_1,\pi_2)\end{array}\right\} \ \delta_1 \geq \delta_2, \pi_1 \leq \pi_2 \tag{68}$$

$$\left.\begin{array}{l}\delta = \delta_2 = \max(\delta_1,\delta_2),\\ \pi = \pi_1 = \max(\pi_1,\pi_2)\end{array}\right\} \ \delta_1 \leq \delta_2, \pi_1 \geq \pi_2 \tag{69}$$

$$\left.\begin{array}{l}\delta = \delta_2 = \max(\delta_1,\delta_2),\\ \pi = \pi_2 = \max(\pi_1,\pi_2)\end{array}\right\} \ \delta_1 \leq \delta_2, \pi_1 \leq \pi_2 \tag{70}$$

However, in the current part of parameter space, constrained by (51), (54), (57) (60) and (62) it is easy to see that only (67) does not conflict with the previously made assumptions. Therefore, the computation only continues for (67), with regard to which, (64) transforms to

$$\tau(A_4,n) = [{a_1}^{\otimes(1+\frac{q}{p})} \otimes a_2 \otimes a_4 \otimes {a_2}^{\otimes(\frac{p}{q}\cdot n)}, a_2 \otimes a_4 \otimes {a_2}^{\otimes(\frac{p}{q}\cdot n)}, \\ a_3 \otimes a_4 \otimes {a_3}^{\otimes(\frac{p}{q}\cdot n)}, {a_4}^{\otimes n}, -\infty, {a_1}^{\otimes(1+\frac{q}{p})} \otimes a_2 \otimes a_4 \otimes {a_2}^{\otimes(\frac{p}{q}\cdot n)}] \otimes \gamma(k). \tag{71}$$

Note that delay-period pairs of the first and the sixth entry of (71) stem from the entries of the dependency vector of a single contribution of (63) that dominates for all $n \in \mathbb{N}_{>0}$ within an iteration. Thus, they are not a conservative approximation but an exact representation of the dependence. The last remaining actor in the quasi-static schedule is A_5, with $\Gamma(A_5) = 1$. As for all actors with a

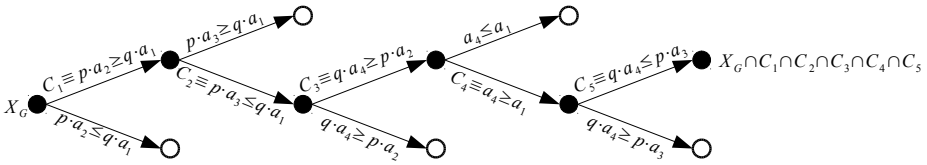


Figure 15: Exploration tree.

constant repetition count, we evaluate (18) for all n up to the repetition vector entry for the actor. We obtain

$$\tau(A_5,1) = \tau(A_4,q) = [{a_1}^{\otimes(1+\frac{q}{p})} \otimes {a_2}^{\otimes(1+p)} \otimes a_4, {a_2}^{\otimes(1+p)} \otimes a_4, \\ {a_3}^{\otimes(1+p)} \otimes a_4, {a_4}^{\otimes q}, -\infty, {a_1}^{\otimes(1+\frac{q}{p})} \otimes {a_2}^{\otimes(1+p)} \otimes a_4] \otimes \gamma(k). \tag{72}$$

The computation of (72) completes the iteration. What remains to be done is to determine the entries of

$$\gamma(k+1) = [t'_{i_1}, t'_{i_2} t'_{i_3}, t'_{i_4}, t'_{i_5}, t'_{i_6}] \tag{73}$$

with regard to the computed iteration in terms of actor responses. For initial tokens placed on actor self-edges it is clear that they are rebuilt after the number of actor firings equal to the entailing actor's repetition count, i.e. repetition vector entry for that actor. Other initial tokens experience a shift along the channel depending on the repetition count of the consuming actor. As these repetition counts are constant (cf. Requirement (1)), so is constant the number of firings of the producing actor due to consistency. Therefore, it is trivial to calculate those shifts. For the example graph

$$t'_1 = \tau(A_1,q), \tag{74} \qquad t'_2 = \tau(A_2,p), \tag{75} \qquad t'_3 = \tau(A_3,p), \tag{76}$$

$$t'_4 = \tau(A_4,q), \tag{77} \qquad t'_5 = \tau(A_5,1), \tag{78} \qquad t'_6 = t_5. \tag{79}$$

Note that token i_5 is not consumed in the current iteration and at the end of the iteration it become token i_6. Thus, (79) holds. If we match the timestamps of (74), (75), (76), (77), (78) and (79) with the producing actor responses (49), (52), (55), (71), (72) and (46), respectively and collect the dependency vectors in a matrix in a row-by-row manner (according to the semantics of the Max-plus scalar product of (16)) we obtain

$$\mathcal{M}_G^{\text{par}}(x^G) = \begin{bmatrix} {a_1}^{\otimes q} & -\infty & -\infty & -\infty & -\infty & {a_1}^{\otimes q} \\ {a_1}^{\otimes(\frac{q}{p}+1)} \otimes {a_2}^{\otimes p} & {a_2}^{\otimes p} & -\infty & -\infty & -\infty & {a_1}^{\otimes(\frac{q}{p}+1)} \otimes {a_2}^{\otimes p} \\ {a_1}^{\otimes(1+q)} \otimes a_3 & -\infty & {a_3}^{\otimes p} & -\infty & -\infty & {a_1}^{\otimes(1+q)} \otimes a_3 \\ {a_1}^{\otimes(1+\frac{q}{p})} \otimes {a_2}^{\otimes(1+p)} \otimes a_4 & {a_2}^{\otimes(1+p)} \otimes a_4 & {a_3}^{\otimes(1+p)} \otimes a_4 & {a_4}^{\otimes q} & -\infty & {a_1}^{\otimes(1+\frac{q}{p})} \otimes {a_2}^{\otimes(1+p)} \otimes a_4 \\ {a_1}^{\otimes(1+\frac{q}{p})} \otimes {a_2}^{\otimes(1+p)} \otimes a_4 & {a_2}^{\otimes(1+p)} \otimes a_4 & {a_3}^{\otimes(1+p)} \otimes a_4 & {a_4}^{\otimes q} & -\infty & {a_1}^{\otimes(1+\frac{q}{p})} \otimes {a_2}^{\otimes(1+p)} \otimes a_4 \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \end{bmatrix} \tag{80}$$

where $x^G \in X_G \cap (p \cdot a_1 \geq q \cdot a_1) \cap (p \cdot a_3 \leq q \cdot a_1) \cap (q \cdot a_4 \leq p \cdot a_2) \cap (a_4 \leq a_4) \cap (q \cdot a_4 \leq p \cdot a_3)$, i.e. x^G belongs to the part of the original graph domain refined by the set of constraints of (51), (54), (57), (60) and (62). Recall that $\mathcal{M}_G^{\text{par}}$ is a mapping that for each $x^G \in X_G$ returns the associated parameterized Max-plus matrix, depending on the affiliation of x^G to a particular partition of X_G, i.e. the parameter space. Furthermore, there exists a finite number of such partitions $X_G = \bigcup_{i=1}^{n} X_{Gi}$ that we call natural SDF-PDFG domains. Each subdomain $X_{Gi} \subseteq X_G$ defines one parameterized matrix. Collected, matrices form the codomain of $\mathcal{M}_G^{\text{par}}$. Fig. 15 illustrates the partitioning of the parameters space (domain) X for the running example. The matrix of (80) is defined by the path determined by the black nodes of the exploration tree.

At this point it is opportune to recall the semantics of an entry of the matrix. In particular, $[\mathcal{M}_G^{\text{par}}(x^G)]_{m,n}$ represents the minimal time distance between token i_m of the $(k+1)$st SDF-PDFG iteration and token i_n of the kth SDF-PDFG iteration. The parametric representation of the matrix elements gives clear insight into the structure of the graph and temporal relationships of actors in the graph. Basically, $[\mathcal{M}_G^{\text{par}}(x^G)]_{m,n}$ defines the latency of the slowest path in the graph connecting

ALGORITHM 1: Compute iteration of an SDF-PDFG.

```
1  Function ExploreGraph(G, T, curr_actor, curr_actor_ndx, curr_in_chan_ndx, curr_init_tok_dep_ndx, curr_init_tok_dep_delay_ndx, curr_init_tok_dep_period_ndx,
   in_contr_comput_completed, constraints, ref res)
2      if (Feasible(constraints) == false) then /* check feasibility of constraints encountered so far                                          */
3          return;
4      end if
5      if (curr_actor == null) then /* pick the next actor from the Qss we have finished with the previous one                                 */
6          curr_actor = G.Qss[curr_actor_ndx];
7      end if
8      if (curr_actor) then /* process actor, by first considering input contributions                                                          */
9          if  (curr_input_chan = curr_actor[curr_in_chan_ndx]) then
10             if ( curr_input_chan[curr_init_tok_dep_ndx]) then
11                 options = compute_output(T[curr_in_chan_ndx][curr_init_tok_dep_ndx], curr_actor.impulse_response);
12                 i = 0;
13                 while (i < options.num_options) do
14                     curr_actor.contributions[curr_in_chan_ndx][curr_init_tok_dep_ndx] = options[i].solution;
15                     ExploreGraph(G, T, curr_actor, curr_actor_ndx, curr_in_chan_ndx, curr_init_tok_dep_ndx + 1, 0, 0, false, constraints +
                       options[i].constraint, res);
16                     i++;
17                 end while
18             else
                   /* completed one input contribution, go to the next                                                                          */
19                 ExploreGraph(G, T, curr_actor, curr_actor_ndx, curr_in_chan_ndx + 1, 0, 0, 0, false, constraints, res);
20             end if
21         else
               /* completed all contributions, proceed with Max-Plus superposition of contributions by combining all delay and period relationships
                  over all initial token dependencies                                                                                           */
22             if (in_contr_completed == false) then
23                 curr_init_tok_dep_ndx = 0;
24                 curr_actor.dp = sort_delays_and_periods(curr_actor.contributions);
25             end if
26             if (curr_actor.dp[curr_init_tok_dep_ndx]) then
27                 if (curr_delay = curr_actor.dp[curr_init_tok_dep_ndx].delays[curr_init_tok_dep_delay_ndx]) then
28                     if (curr_period = curr_actor.dp[curr_init_tok_dep_ndx].periods[curr_init_tok_dep_period_ndx]) then
29                         T[curr_actor_output_chan_ndx_all][curr_init_tok_dep_ndx].delay = curr_delay.value;
30                         T[curr_actor_output_chan_ndx_all][curr_init_tok_dep_ndx].period = curr_period.value;
                           /* next period for current delay                                                                                    */
31                         ExploreGraph(G, T, curr_actor, curr_actor_ndx, 0, curr_init_tok_dep_ndx, curr_init_tok_dep_delay_ndx,
                           curr_init_tok_dep_period_ndx + 1, true, constraints + options[i].constraint);
32                     else
                           /* next delay                                                                                                       */
33                         ExploreGraph(G, T, curr_actor_ndx, 0, 0, curr_init_tok_dep_ndx, 0, curr_init_tok_dep_delay_ndx + 1, true, constraints,
                           res);
34                     end if
35                 else
                       /* next initial token dependency                                                                                        */
36                     ExploreGraph(G, T, curr_actor, curr_actor_ndx, 0, curr_init_tok_dep_ndx + 1, 0, 0, true, constraints, res);
37                 end if
38             else
                   /* done with this actor, do the next one                                                                                    */
39                 ExploreGraph(G, T, null, curr_actor_ndx + 1, 0, 0, 0, 0, false, constraints, res);
40             end if
41         end if
42     else
           /* no more actors in the qss, this is a leaf node - build the matrix associated with a set of constraints                           */
43         res += process(G,T,constraints);
44     end if
45     return;
46 end
```

two initial tokens. This path is determined by the delay that all actors along the path contribute to and by the period of the slowest actor in the path. E.g. if we consider $[\mathcal{M}_G^{\text{par}}(x^G)]_{4,1}$ as obtained from (71), we see that actor A_2 is the bottleneck of the path from i_1 to i_4, i.e. it has the highest period (or in the light of conventional linear system theory it has the lowest cutoff frequency). On the other hand from a concrete Max-plus matrix obtained by the procedure of Section 3.3 such relationships cannot be studied.

6.2.4 Computation of the SDF-PDFG iteration

Algorithm 1 specifies the previously described procedure for the computation of one iteration of a SDF-PDFG.

It is defined by a recursive function `ExploreGraph` that explores the tree-like structures like that of Fig. 15 in a depth first search manner. The inputs to the function are `G` the SDF-PDFG itself with all associated meta-data like the quasi-static schedule of the structure, `T` the set of dependency vectors of all graph channels, `curr_actor` the structure containing all required meta-data for the actor being currently evaluated, `curr_actor_ndx` the index of the current actor under processing in the quasi-static schedule, `curr_in_chan_ndx` the index of the currently processed input of the currently processed actor, `curr_init_tok_dep_ndx` the index of the currently processed entry of the dependency vector either within a Max-plus convolution or Max-plus superposition context, `curr_init_tok_dep_delay_ndx` the index of the currently set maximal delay among all the delays

observed for the currently processed dependency vector entry, `curr_init_tok_dep_period_ndx` the index of the currently set maximal period among all the periods observed for the currently processed dependency vector entry, `in_contr_comput_completed` the flag denoting whether or not all input channel contributions have been considered for the currently processed actor, `constraints` the set of constraints defining the parameter space partition of the current exploration path and `res` (passed by reference) the result set containing parameterized matrices governing the behavior of the SDF-PDFG in a partition of the initial domain defined by all the constraints encountered along the exploration path.

In the function, actors are processed as ordered in the quasi-static schedule (cf. Line 6). Once the last actor had been processed, the dependency vectors are composed into the related parameterized matrix and along with the constraints encountered added to the result set (cf. Line 43). In the processing of a particular actor the contributions stemming from all its input channels are processed one by one (cf. Line 9). This may incur as many Max-plus convolutions as there are entries in the input dependency vector (cf. Line 10). As a convolution (cf. Line 11) incurs splitting of the parameter space, the search is recursively continued for each splitting option (cf. Line 15) while proceeding with the next dependency vector entry (note the increment of `curr_init_tok_dep_ndx` in the recursive call). Note that there are maximally two options per dependency vector entry. Of course, newly added constraints should not conflict with the previous ones, i.e. their feasibility needs to be verified (cf. Line 2). If the combination is not feasible this branch of exploration is left all together (cf. Line 3). Once all entries of the dependency vector of one input channel have been processed, the algorithm continues with the next input channel (cf. Line 19).

Once the contributions of all inputs have been computed, the algorithm performs the bounding of delay-period pairs over all corresponding entries of dependency vectors of different contributions according to Proposition 2. This is marked by resetting `curr_init_tok_dep_ndx` if the flag `in_contr_comput_completed` is set to `false` to denote that the actor had not yet been treated by Proposition 2. Note that in later recursive calls, the flag `in_contr_comput_completed` will be set to `true` (cf. Lines 31, 33 and 36). All combinations are considered, i.e. for each dependency vector entry (cf. Line 26) a different pivot delay (cf. Line 29) and a pivot period (cf. Line (30)) are set and the search is continued for the next period that is to be deemed maximal (cf. Line 31). Once all periods have been exhausted (cf. Line 28), we proceed with the next pivot delay (cf. Line 33). Once all delays have been exhausted, we proceed with the next dependency vector entry (cf. Line 36). Note that after every recursive call the feasibility is verified with the newly added constraint. The search is aborted in the current branch if the feasibility check fails (cf. Line 3). Once ending with the current actor, we proceed to the next one (cf. Line 39) until the quasi-static schedule has been entirely processed and the matrix added to the solution set (cf. Line 43) . The algorithm has exponential time complexity in the worst case. However, practical application may be expected to have only a few critical parameters while the graph structures can expose sparsity in the sense that there will be no dependencies between many initial tokens in the graph. Furthermore, the definitions of the domains may be such that many exploration paths will be pruned out due to infeasibility.

The set of matrices obtained via Algorithm 1 when evaluated at a corresponding $x^G \in X_G$ are actually conservative approximations of the corresponding Max-plus instance matrices. This is due to the conservativity entailed by Propositions (1) and 2. Therefore, the following inequality holds

$$\left[\mathcal{M}_G^{\text{par}}(x^G)\right](x^G) \geq \mathcal{M}_G(x^G) = M_{\iota_G(x^G)} \tag{81}$$

for all $x^G \in X_G$. The approximation of (81) is tight as per matrix entry it only incurs an added element of delay while the period of the sequence used to obtain the actual matrix value is exact and captures the slowest actor in the path between two tokens (cf. Propositions (1) and 2). We show this by example. E.g., we mentioned in the prelude that when we evaluate $G = s_1^p$ SDF-PDFG of Fig. 6a at the configuration $x^G = \{p = 3, q = 2, a_1 = 5, a_2 = 4, a_3 = 3, a_4 = 4\}$ its instance SDFG emerges, denoted $\iota_G(x^G)$. In particular, this instance corresponds to the SDFG of Fig. 2 whose Max-plus matrix is given by (7). We consider an arbitrary element of the two matrices. With (80), $\left[(\mathcal{M}_G^{\text{par}}(x^G))(x^G)\right]_{4,1} = 28.3$, while with (7), $[M_{\iota_G(x^G)}]_{4,1} = 22$. For growing values of p and q, i.e. for the growing repetition vector the period component becomes dominant and the relative error shrinks.

E.g, with $x^G = \{p = 300, q = 200, a_1 = 5, a_2 = 4, a_3 = 3, a_4 = 4\}$, $\left[\left(\mathcal{M}_G^{\text{par}}(x^G)\right)(x^G)\right]_{4,1} = 1216.3$, while $[M_{\iota_G(x^G)}]_{4,1} = 1210$.

7 Worst-case performance analysis of parameterized synchronous dataflow scenarios

In this section we investigate the performance analysis problem for SDF-PFSM-SADF specifications. In particular, we are interested in deriving tight worst-case throughput and latency estimates. The problem at hand is challenging due to several reasons.

First, SDF-PFSM-SADF is a dynamic dataflow model able to express data dependent behavior within its concurrency model called SDF-PDF and intricate control logic captured at the level of the enveloping FSM. Therefore, its temporal behavior is from an observer's point of view is irregular, i.e. it show no periodicity properties like that of, for instance, SDF. Thus, simple approaches to analysis not accounting for scenario transitions such as analyzing the worst-case scenario (the one with the lowest throughput and highest) or deriving a SDF-PDFG from all scenario SDF-PDFGs by taking the worst-case firing delays over all parameterized scenarios and subjecting it to analysis may not provide conservative and tight bounds [19].

Second, the model's execution progresses at a scenario granularity where scenarios may be concurrently active, i.e. pipelined.

Third, consecutive scenarios are inter-dependent and require synchronization. This synchronization is achieved by the use of initial tokens that exist in between scenarios.

7.1 Max-plus albebraic semantics of SDF-PFSM-SADF

By Definition (6) an SDF-PFSM-SADF is determined by a set of parameterized scenarios S^p and an FSM on that set. Every parameterized scenarios $s_j^p \in S^p$ is represented by an SDF-PDFG and an associated domain of the scenario SDF-PDFG $X_{s_j^p}$. Let

$$\mathsf{M} = \{\mathcal{M}_{s_1^p}^{\text{par}}, \ldots, \mathcal{M}_{|S^p|}^{\text{par}}\} \tag{82}$$

be the set of all mappings of that per particular scenario given the scenario SDF-PDFG configuration return the associated parameterized scenario matrix (recall the semantics of $\mathcal{M}_G^{\text{par}}$ from (14)). Now, the operational semantics of SDF-PFSM-SADF says that an SDF-PFSM-SADFG evolves in iterations of its scenario SDF-PDFG instances where the scenario occurrence patterns is given by the scenario FSM. Scenarios are synchronized the set of initial tokens whose production times are in turn captured by the initial token timestamp vectors.

Therefore, the timestamp vectors of initial tokens after the $(k+1)$st iteration can be related to the timestamp vector of initial tokens after the kth iteration as follows

$$\gamma(k+1) = \underbrace{\left[(\mathbb{M}(\pi_l(\zeta(k+1))))(\pi_r(\zeta(k+1)))\right](\pi_r(\zeta(k+1))}_{M_{\iota_{\pi_l(\zeta(k+1))}(\pi_r(\zeta(k+1)))}} \otimes \gamma(k). \tag{83}$$

In (83), mapping $\zeta : \mathbb{N}_{>0} \to (S^p \times X)$ returns the active scenario of the $(k+1)$st SDF-PFSM-SADFG iteration as well as its configuration from the set $X = X_{s_1^p} \cup \ldots \cup X_{s_{|S^p|}^p}$, while $\mathbb{M} : S^p \to \mathsf{M}$ returns the configuration to parameterized Max-plus matrix mapping of a particular scenario and π_l and π_r are the left and right projection functions, respectively. Finally, the matrix of the underbrace of (83) is the Max-plus matrix of the SDFG instance running as the $(k+1)$st iteration of the SDF-PFSM-SADFG. To explain the cumbersome notation of (83), we use (84).

$$\begin{aligned}(k+1) \xrightarrow{\zeta(k+1)} (s_j^p \in S^p, x^{s_j^p} \in X_{s_j^p}) \xrightarrow{\mathbb{M}(s_j^p)} \mathcal{M}_{s_j^p}^{\text{par}} \xrightarrow{\mathcal{M}_{s_j^p}^{\text{par}}(x^{s_j^p})} [\mathcal{M}_{s_j^p}^{\text{par}}(x^{s_j^p})] \\ \xrightarrow{[\mathcal{M}_{s_j^p}^{\text{par}}(x^{s_j^p})](x^{s_j^p})} M_{\iota_{\pi_l(\zeta(k+1))}(\pi_r(\zeta(k+1)))}\end{aligned} \tag{84}$$

Starting from the SDF-PFSM-SADF iteration index $(k+1)$ by applying ζ, the running parameterized scenario and its running configuration are determined as a pair $(s_j^p \in S^p, x^{s_j^p} \in X_{s_j^p})$. Thereafter s_j^p is used as an argument to $\mathbb{M}$ to determine the mapping from configurations to parameterized matrices within the scenario (cf. (82)), i.e. $\mathcal{M}^{\text{par}}_{s_j^p}$. After the mapping had been determined, it is used to determine the parameterized matrix corresponding to the configuration $x^{s_j^p}$ from the initial pair. This matrix is denoted $[\mathcal{M}^{\text{par}} s_j^p(x^{s_j^p})]$. Finally, this matrix is evaluated at $x^{s_j^p}$ which gives us the conservative approximation of Max-plus matrix of the running scenario SDFG instance, denoted $M_{\iota_{\pi_l(\zeta(k+1))}(\pi_r(\zeta(k+1)))}$.

The recursion of (83) can be used to define the completion time of a sequence of parameterized scenarios

$$\sigma = s_1^p, \ldots, s_k^p \tag{85}$$

that we consider a part of the scenario FSM trace. As an activation of the scenario entails the activation of an arbitrary scenario SDFG instance (nondeterministic choice), we expand (85) as follows

$$\sigma = (s_1^p, x_1^{s_1^p} \mid \ldots \mid x_{|X_{s_1^p}|}^{s_1^p}), \ldots, (s_k^p, x_1^{s_k^p} \mid \ldots \mid x_{|X_{s_k^p}|}^{s_k^p}), \tag{86}$$

where bar $\mid$ denotes a nondeterministic choice. Then, the completion time of (86) can be defined as follows

$$\mathcal{A}^p(\sigma) = \alpha^{pT} \otimes \mu^p(\sigma) \otimes \beta^p \tag{87}$$

where α^p is the final delay, β^p is the initial delay and $\mu^p : S^{p*} \to \mathbb{R}_{\max}^{I \times I}$ is a morphism that associates sequences of scenarios σ with Max-plus matrices as follows

$$\begin{aligned}
\mu^p(\sigma) &= \mu^p((s_1^p, x_1^{s_1^p} \mid \ldots \mid x_{|X_{s_1^p}|}^{s_1^p}), \ldots, (s_k^p, x_1^{s_k^p} \mid \ldots \mid x_{|X_{s_k^p}|}^{s_k^p})) \\
&= \left[(\mathbb{M}(s_1^p)) \, (x_1^{s_k^p} \mid \ldots \mid x_{|X_{s_k^p}|}^{s_k^p}) \right] (x_1^{s_k^p} \mid \ldots \mid x_{|X_{s_k^p}|}^{s_k^p}) \otimes \ldots \\
&\otimes \left[(\mathbb{M}(s_1^p)) \, (x_1^{s_1^p} \mid \ldots \mid x_{|X_{s_1^p}|}^{s_1^p}) \right] (x_1^{s_1^p} \mid \ldots \mid x_{|X_{s_1^p}|}^{s_1^p})
\end{aligned} \tag{88}$$

The triple $\mathcal{A}^p = (\alpha^p, \mu^p, \beta^p)$ defines the Max-plus automaton structure of [16]. In (87), β^p captures the initial enabling times of graph's initial tokens, i.e. $\beta^p = \gamma(0)$ and typically $\gamma(0) = \mathbf{0}$. On the other hand α^p specifies the metrics we are interested in. E.g., if we are interested in the makespan of the scenario sequence, we set $\alpha^p = \mathbf{0}$.

The Max-plus automaton structure of (87) with (88) can be used to study the performance of SDF-PFSM-SADF in a similar fashion as it had been used to study the performance of FSM-SADF. By comparing the Max-plus automata structure of FSM-SADF (cf. (9) and (10)) and SDF-PFSM-SADF (cf. (87) and (88)) we observe a striking resemblance. Both in FSM-SADF and PFSM-SADF the Max-plus automata structures are defined on a finite number of scenarios. However the crucial difference lies in the definition of automata morphisms of (10) and (88). In FSM-SADF, the Max-plus automata morphism μ returns for a scenario the corresponding scenario SDFG matrix and for each scenario there is only one such matrix. With SDF-PFSM-SADF, the mapping μ^p for a parameterized scenario returns the Max-plus matrix of an arbitrarily chosen parameterized scenario SDFG instance. Further comparison of the morphisms reveals that (88) can be unfolded into (10) in a way that every parameterized scenario instance would become a scenario in an equivalent FSM-SADF. This follows straight forwardly from the discussion on operational semantics of PFSM-SADF compared to that of FSM-SADF of Section 5. Then the equivalent structure could be used to analyze the original parameterized specification for worst-case performance. However, in practice, this is not feasible because Max-plus automata-based techniques for throughput analysis of FSM-SADF rely on the automata product structure which would explode in size due to unfolding. The same effect would incapacitate the use of state-space-based latency analysis techniques. This does not mean we actually give up on the FSM-SADF techniques, but we only need to find a way

ALGORITHM 2: Worst-case evaluation of a parameterized scenario.

Data: Mapping $\mathbb{M}(s_k^p)$

Result: Worst-case evaluation matrix of a parameterized scenario $M_{s_k^p}^{\mathrm{w-c}}$

1 **for** $i = 1$ **to** $|I|$ **do**
2 **for** $j = 1$ **to** $|I|$ **do**
3 $[M_{s_k^p}^{\mathrm{w-c}}]_{i,j} = -\infty$;
4 **foreach** $M^{\mathrm{par}} \in cod(\mathbb{M}(s_k^p))$ **do**
5 **if** $[M^{\mathrm{par}}]_{i,j} \neq -\infty$ **then**
6

$$tmp = \underset{x}{\text{maximize}} \quad [M^{\mathrm{par}}(x)]_{i,j} \qquad \text{subject to} \quad x \in dom(\mathbb{M}(s_k^p)) \text{ s.t. } \mathbb{M}(s_k^p)(x) = M^{\mathrm{par}} \tag{92}$$

$$[M_{s_k^p}^{\mathrm{w-c}}]_{i,j} = [M_{s_k^p}^{\mathrm{w-c}}]_{i,j} \oplus tmp$$

7 **end if**
8 **end foreach**
9 **end for**
10 **end for**

how to compact the representation of (88), i.e. remove the need for explicit consideration of all nondeterministic choices. As we are interested in the worst-case performance metrics, the theory of Max-plus automata provides the solution to the problem.

In particular, every element of the product in (88) is replaced by its worst-case evaluation according to Theorem 2 of [16] as follows

$$\left[(\mathbb{M}(s_k^p))\,(x_1^{s_k^p} \mid \ldots \mid x_{|X_{s_k^p}|}^{s_k^p})\right](x_1^{s_k^p} \mid \ldots \mid x_{|X_{s_k^p}|}^{s_k^p}) = \bigoplus_{x_i^{s_k^p} \in X_{s_k^p}} \left[(\mathbb{M}(s_k^p))(x_i^{s_k^p})\right](x_i^{s_k^p}), \tag{89}$$

$$M_{s_k^p}^{\mathrm{w-c}} = \bigoplus_{x_i^{s_k^p} \in X_{s_k^p}} \left[\mathbb{M}(s_k^p)(x_i^{s_k^p})\right](x_i^{s_k^p}). \tag{90}$$

In, (90), we call $M_{s_k^p}^{\mathrm{w-c}}$ the worst-case evaluation matrix of a parameterized scenario. With (90), (88) transforms to

$$\mu^p(\sigma) = M_{s_k^p}^{\mathrm{w-c}} \otimes \ldots \otimes M_{s_1^p}^{\mathrm{w-c}}. \tag{91}$$

The actual $M_{s_k^p}^{\mathrm{w-c}}$ of (90) per parameterized scenario can be obtained by solving a sequence of optimization problems specified by Algorithm 2. The input to the algorithm is the mapping $\mathbb{M}(s_k^p)$, while the output is the desired $M_{s_k^p}^{\mathrm{w-c}}$ of (90). We use the notation *dom* and *cod* for the domain and codomain of a mapping, respectively. Each entry of $M_{s_k^p}^{\mathrm{w-c}}$ corresponds to the maximal entry among all corresponding maximal entries of parameterized scenario matrices defining the codomain of $\mathbb{M}(s_k^p)$ (cf. Line 4). These entries on the other hand correspond to the maximum value an entry of the parameterized matrix attains when evaluated for all configurations within the subdomain the matrix is defined in. It is obtained by solving the optimization problem of (92) where the objective function is the entry of the considered parameterized matrix, i.e. a parameterized expression. Recall that Algorithm 1 returns a set of parameterized matrices valid in different subregions of the scenario domain. These subregions we called natural scenario subdomains.

The type of optimization problems encountered in (92) depends on the formulations of $\mathcal{R}$ and $\mathcal{D}$ in the definition of the parameterized scenario (cf. Definition 4) as well as on the specification of the scenario domain. With the formulations of (11) and (12) and with regard to Propositions 1 and 2 and the discussion of Section 6.2, the objective function of (92) will fall into the class of polynomial programming problems. Techniques exist for producing global solutions to such problems which is

of crucial importance as we are to give worst-case guarantees. We refer to [34] for more details on how to solve polynomial programming problems. The global solvability poses a vital technical constraint. When it comes to the definition of the scenario domain that with the definitions of $\mathcal{R}$ and $\mathcal{D}$ determines the type of the optimization problems we encounter, it is the designer's responsibility to specify the domain in a way that a global solver for the problem of (92) exists.

7.2 Performance metrics for SDF-PFSM-SADF

Now, given an SDF-PFSM-SADFG with the associated Max-plus automaton with μ^p as represented by (91), we proceed with throughput analysis. First, we define this metric of interest.

Definition 8 (Throughput)**.** *Throughput of an PFSM-SADFG is defined as the largest value $\rho \in \mathbb{R}$ s.t. for every possible sequence of PDFG scenarios to which we associate the corresponding timestamp vector sequence vector sequence $\overline{\gamma}$, for every $\epsilon \in \mathbb{R}$ s.t. $\epsilon > 0$, there is some $K \in \mathbb{N}_{>0}$ s.t. for all $L \in \mathbb{N}_{>0}$, $L > K$, $\frac{L}{||\overline{\gamma}(L)||} > \rho - \epsilon$.*

The throughput is defined in terms of number of iterations per time unit in correspondence with the definition of throughput used for SDF [21] and FSM-SADF [19][36]. Although, it would be more intuitive to define the throughput as the limit of the long-run average number of completed iterations per time unit, this definition would not suffice as this long-average may not exist for all scenario sequences. In reality, it may bounce between superior and inferior limiting bounds [19][36]. Therefore, we have adhered to a somewhat cumbersome Definition 8 adopted from [19].

As it may be, the worst-case throughput of an SDF-PFSM-SADF specification can be determined by using the associated Max-plus automata triple restricted to a regular sublanguage of S^{p*} defined by the parameterized scenario FSM similarly as done for FSM-SADF in [19]. We adopt the context to that of SDF-PFSM-SADF. In particular, the worst-case throughput of an SDF-PFSM-SADFG is equal to the maximum cycle mean (MCM) of the so called throughput graph of that SDF-PFSM-SADFG. In parlance of Max-plus automata theory, the throughput graph corresponds to the communication graph of the maximal matrix among the tensor product matrices of particular worst-case evaluation matrices of (90). For more details we refer to [16]. However, these products need not to be computed and the throughput graph can be constructed directly using the worst-case evaluation matrices and the scenario FSM as follows.

Traverse over all scenario FSM states $\phi_u^p \in \Phi^p$ and add a node to the throughput graph for each initial token $i_v \in I$ of the SDF-PFSM-SADFG and label the node with (ϕ_u^p, i_v). Then for every transition (ϕ_r^p, ϕ_s^p) add an edge from node (ϕ_r^p, i_m) to node (ϕ_s^p, i_n) if $[M^{\mathrm{w-c}}_{\Psi^p(\phi_s^p)}]_{n,m} \neq -\infty$ and set the weight of the edge to $[M^{\mathrm{w-c}}_{\Psi^p(\phi_s^p)}]_{n,m}$.

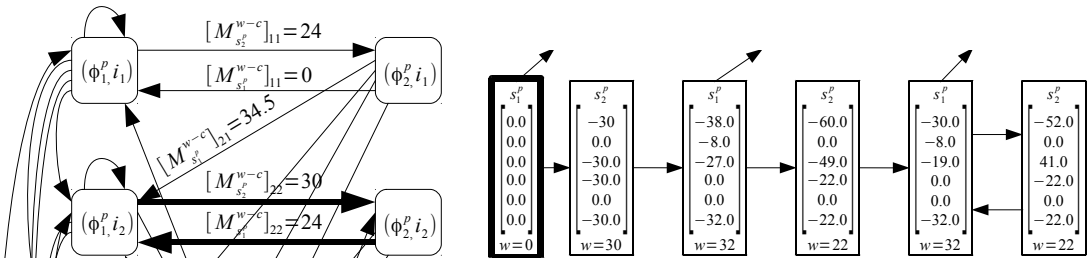
These weights represent minimal distances between initial tokens between consecutive scenarios. Over infinite sequences of scenarios these distances will be part of the throughput graph cycles. Therefore, the inverse of the MCM of the throughput graph defines the worst-case throughput value.

We exemplify using the running example SDF-PFSM-SADF of Fig. 6. Assume that the respective scenario SDF-PDFG domains are given as follows

$$
\begin{aligned}
X_{s_1^p} = & C_1 \cap C_2 \cap C_3 \cap C_4 \cap C_5 \cap \\
& \big\{ p = w_1 \cdot w_2, w_1 + w_2 = 2 \cdot x_1 - x_2, \\
& p \in [1,10], q \in [1,10], w_1 \in [1,3], w_2 \in [1,4], \\
& x_1 \in [1,3], x_2 \in [1,5], a_1 \in [1,7], a_2 = 4, \\
& a_3 \in [1,5], a_4 = 4 \big\}
\end{aligned} \tag{93}
$$

$$
X_{s_2^p} = \{ u = 30 \} \tag{94}
$$

Equation (93) is a very illustrative example of a domain specification because it shows how graph parameters (rates and actor firing delays) may exhibit arbitrary dependence on parameters not present in the graph itself in a nested fashion. E.g. parameterized rate p nonlinearly depends on parameters w_1 and w_2 which in turn depend on parameters x_1 and x_2. The domain (93) in addition



(a) Throughput graph. (b) State-space.

Figure 16: Performance analysis structures.

defines a default parameter interval for each parameter, e.g $p \in [1, 10]$. For illustration purposes, in (93) we assume that the scenario SDF-PDFG domain is a subset of the natural scenario subdomain defined by the constraints encountered while producing the matrix of (80). This way, during the generation of $M^{\mathrm{w-c}}_{s_1^p}$, Algorithm 2 needs only to maximize over the entries of (80). The domain of s_2^p encloses only one configuration specified by (94). After running Algorithm 2, we obtain the worst-case evaluation matrices of the respective parameterized scenarios as specified by (95).

$$M^{\mathrm{w-c}}_{s_1^p} = \begin{bmatrix} 24 & -\infty & -\infty & -\infty & -\infty & 24 \\ 34.5 & 24 & -\infty & -\infty & -\infty & 34.5 \\ 34 & -\infty & 24 & -\infty & -\infty & 34 \\ 42.5 & 32 & 32 & 24 & -\infty & 42.5 \\ 42.5 & 32 & 32 & 24 & -\infty & 42.5 \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \end{bmatrix} \quad M^{\mathrm{w-c}}_{s_2^p} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\ -\infty & 30 & -\infty & -\infty & -\infty & 30 \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & 0 & -\infty & -\infty \\ -\infty & 30 & -\infty & -\infty & -\infty & 30 \\ -\infty & -\infty & -\infty & -\infty & 0 & -\infty \end{bmatrix} \tag{95}$$

Using the matrices of (95) we now construct the throughput graph which is partially displayed in Fig. 16a. The critical cycle of the throughput graph is depicted using bold arrows. The critical cycle defines the MCM of the throughput graph which attains the value $(30+24)/2 = 27$. The worst-case throughput ρ equals to the inverse of the MCM, i.e. $\rho = 1/27$ iterations per time-unit.

At this point one might argue, that given the relatively small cardinality of the scenario domains of (93) one could enumerate the domains and use techniques of FSM-SADF to obtain the worst-case throughput value. This claim holds for the running example, but in practice the respective domains can be vast and enumeration infeasible. Another might argue that instead of computing the worst-case evaluation matrices of parameterized scenarios via Algorithms 2 and 1, one could simply construct the worst-case SDFG of a parameterized scenario graph by taking the upper endpoints of default parameter intervals which would in turn define a worst-case FSM-SADF that can be analyzed. In response to this, we argue that such an approach (although straightforward) which disregards complex parameter dependencies defined by the parameterized scenario domain specifications might incur to much pessimism. For the running example, by merely taking $p = 10$, $q = 10$ and $u = 30$ with $a_1 = 7$, $a_2 = 4$, $a_3 = 5$ and $a_4 = 4$ we obtain an FSM-SADFG with the worst-case throughput value of $1/70$ iterations per time unit which is an over-approximation of the actual throughput value of $1/24$ iterations per time-unit.

The remaining performance metric to be discussed is latency. As for throughput we adopt the definition from [19].

Definition 9 (Latency)**.** *Latency of an PFSM-SADFG relative to the desired period $\pi \in \mathbb{R}$ is defined as the smallest vector λ such that for every timestamp vector sequence $\overline{\gamma}$, for every $k \geq 0$, $\overline{\gamma}(k) \leq k \cdot \pi + \lambda$.*

Using the worst-case evaluation matrices of the parameterized scenarios along with the scenario FSM we obtain latency estimates via the analysis of the reachable part of the state space of all timestamp vectors $\gamma(k)$. State space is constructed in a bread-first search manner from the parameterized scenario FSM. The state itself is defined as a tuple $(\Psi^p(\phi^p), \gamma, w)$ where $\phi^p \in \Phi^p$, γ is

a Max-plus timestamp vector which is used to initialize the next scenario execution and w is the state weight. Let tuple $(\Psi^p(\phi^{p\prime}), \gamma', w')$ define a state that is directly reachable from $(\Psi^p(\phi^p), \gamma, w)$. In that case, $\gamma' = (M^{\mathrm{w-c}}_{\Psi^p(\phi^{p\prime})} \otimes \gamma)^{\mathrm{norm}}$ and $w' = ||M^{\mathrm{w-c}}_{\Psi^p(\phi^{p\prime})} \otimes \gamma||$. Continuation of the state-space construction will eventually result in revisiting an already existing state if the reachable part of the state space is finite. The exploration terminates, when there are no more new states. For any path of length k leading to state $(\Psi^p(\phi^p), \gamma, w)$, the actual $\gamma(k)$ of the associated parameterized scenario sequence is given by $T \otimes \gamma$ where T equals to the sum of the weights of the path states. Assuming we know the throughput of the graph, we can determine the latency in a single traversal of state space by finding the smallest vector λ such that $\gamma(k) \leq \lambda + \frac{k}{\rho}$. This equals to determining the maximal value of $\gamma(k) - \frac{k}{\rho}$ observed. The exploration needs to consider only acyclic paths in the state space as any cycle will not be faster than determined by the throughput. We demonstrate this for the running example in (96) over the state space path of Fig. 16b.

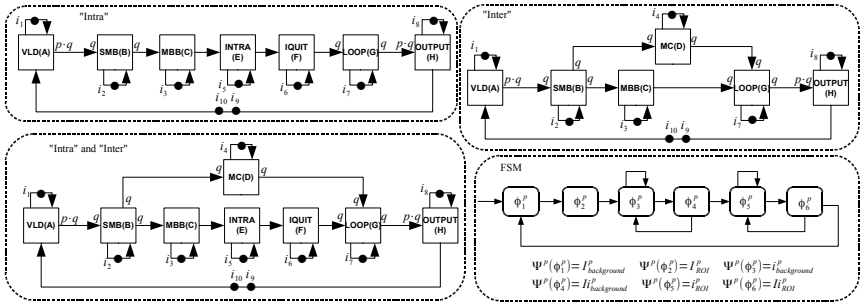
$$\begin{aligned}\lambda = \bigoplus \big\{ & [0,0,0,0,0,0], [0,30,0,0,30,0] - 27, \\ & [24,54,35,62,62,30] - 54, [24,84,35,62,84,62] - 81, \\ & [86,108,97,116,116,84] - 108, \\ & [86,138,97,116,138,116] - 135 \big\} = [0,3,0,8,8,0]\end{aligned} \tag{96}$$

Paper [19] in Proposition 4.1 gives a practical condition under which the reachable part of the state space is finite. It says (recast in the context of PFSM-SADF) that for every possible scenario sequence σ allowed by the FSM and any $k > 0$ there is some $m > 0$ such that the matrix $\mu(\sigma)$ contains no entries $-\infty$. We argue that this condition is too restricting. Imagine a PFSM-SADF/FSM-SADF with only one scenario defined by an actor with two initial token in the feedback loop. The Max-plus representation of such a scenario is a 2 by 2 Max-plus matrix with $-\infty$ on the diagonal. And therefore, any power of this matrix has entries $-\infty$ on the diagonal too and this structure cannon satisfy the finiteness conditions of [19] although the state-space of such a specification is finite (we leave it to the interested reader as a small exercise to generate the state space of this simple structure). We give a new and less restricting condition for fitness of the reachable part of the state space by following the observation of [36] that any scenario sequence can be formed by concatenation of cycles of the scenario FSM. First we briefly elaborate on the concept of irreducibility in Max-plus. A matrix $M \in \mathbb{R}^{n\times n}_{\max}$ is called irreducible if its communication graph $\mathcal{G}(M)$ is strongly connected [26]. The communication graph of $M \in \mathbb{R}^{n\times n}_{\max}$, denoted $\mathcal{G}(M)$, is a graph with the set of nodes given by $\mathcal{N}(M) = \{1, \ldots, n\}$ where a pair $(i,j) \in \mathcal{N}(M) \times \mathcal{N}(M)$ is an edge of the graph if $m_{j,i} \neq -\infty$. For more details we refer to [26].

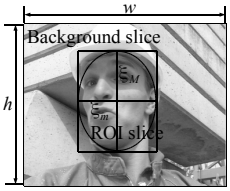
Proposition 3. *Let $\mathsf{F}^p = (S^p, F^p)$ be an SDF-PFSM-SADFG. Let $C = \{c_i\}$ by the set of all simple cycles of F^p. If the matrix $M_{c_i} = \bigoplus_{n=1}^{length(c_i)} M_{\Psi^p(c_i(n))}$ is irreducible for every $c_i \in C$ where $M_{\Psi^p(c_i(n))}$ is the matrix of an arbitrary instance of scenario $\Psi^p(c_i(n))$, then the reachable part of the state space is finite.*

Proof. Irreducibility of a Max-plus matrix M_{c_i} implies that its eigenvalue is unique [26]. The eigenvalue on the other hand specifies the asymptotic growth rate of a timestamp vector produced by this matrix (cf. (4)). Therefore, entries of the normalized timestamp vectors generated by M can only take values from a bounded range because the growth rate is the same for all the entries. Therefore, in consideration of a scenario sequence as a repetitive pattern consisting only of one FSM cycle, there will be a finite number of timestamp vectors within the sequence which implies the finiteness of the state space over one cycle. The argument straightforwardly carries over to concatenations of different cycles as they are all individually bounded, i.e. no token timestamp can diverge. □

Note that to check for finiteness of the state space, it is enough to take the Max-plus matrix of arbitrarily chosen parameterized scenario instance as the irreducibility criteria only considers the structure of the matrix (e.g. sparsity), and not the values of particular entries. If we verify the example with an actor with two initial tokens in a feedback loop where the finiteness criteria of [19]



(a) VC-1 decoder captured in SDF-PFSM-SADF.



(b) ROI coding.

Figure 17: Case study.

fails to give an answer against Proposition 3, we see that Proposition 3 gives a positive answer to the question, which is the correct one.

8 Case study

In this section, we demonstrate the application of our parameterized scenario modeling and analysis techniques to a realistic case study from the multimedia domain. In particular, we consider the case of a VC-1 video decoder used in a region of interest (ROI) coding scheme.

ROI coding [23] is a feature of modern video codecs that allows to independently store and transmit a video in a variety of regions of interest. This feature is useful for achieving higher error resilience as errors cannot cross ROI boundaries or for saving bandwidth as a ROI can be coded with more bits to obtain a much higher-quality than that of the non-ROI which is coded with fewer bits. Typical way of representing ROIs in a video picture is by the use of a rectangular region that corresponds to a picture slice. Slice is a group of macroblocks. The rectangular mapping of ROI macroblocks to slices corresponds to the foreground mapping type defined within the flexible macroblock ordering scheme (FMO), know as FMO Type 2 [23]. We exemplify using the the picture from the *Foreman* sequence shown in Fig. 17b. In the sequence the region of interest is the foreman's face represented by the rectangular "ROI slice", while the background is represented by the "Background slice". To achieve this representation, at the encoder side the video is preprocessed by a face detection algorithm, which gives the relative position of the foreman's face. Thereafter, the face and the background regions are embodied into separate slices, encoded (possibly by using different quality settings) and stored/transmitted.

In VC-1 coding, three different types of slices are supported: I, i and Ii slices. In an I-slice all macroblocks are encoded in the *Intra* mode. In an i-slice all macroblocks are encoded in the *Inter* mode. In an Ii-slice all macroblocks are both *Intra* and *Inter* coded. The types of slices naturally represent three modes of operation of the decoder shown in Fig. 17a adopted from [4]. Each mode is represented by a different SDF-PDFG according to our parameterized scenario modeling technique. Each SDF-PDFG iteration corresponds to decoding of one slice. Actor *VLD* implements the variable length decoder, actor *SMB* splits macroblocks into blocks, actor *MBB* splits macroblocks into blocks, actor *INTRA* performs intra decoding at the block level, *IQUIT* implements inverse integer transform at the block level, actor *LOOP* implements the deblocking filter at the macrblock level, actor *MC* performs motion compensation at the macroblock and actor *OUTPUT* stores the decoded slice into the output frame buffer. Graph rates p and q denote the number of macroblocks in a slice and the number of blocks within a macroblock, respectively. Actor execution times (not displayed) are adopted from the profiling results of [4].

We proceed in the context of face detection of Fig. 17b where the decoder's task is to decode a sequence of images where each image is split into two slices, one being the ROI (foreman's face) and the other being non-ROI (background). With two slice types amenable to processing (ROI and background) and in consideration of the nature of the slices (I, i and Ii), we obtain six decoder

scenarios: I^p_{ROI}, i^p_{ROI}, Ii^p_{ROI}, $I^p_{\text{background}}$, $i^p_{\text{background}}$ and $Ii^p_{\text{background}}$. E.g., scenario I^p_{ROI} models the decoding of an I slice capturing the ROI, i.e. the foreman's face. Relative to the given input frame resolution, the slice sizes expressed in macrobolocks (parameterized rate p) will differ at runtime, depending on the distance of the foreman's face from the capturing device, i.e. camera.

We assume the ROI can be abstracted into an ellipse of known characteristics, i.e. of known circumference o and eccentricity ϵ where ξ_M and ξ_m are the major and minor axis of the ellipse, respectively. The ellipse abstraction is a natural representation for a face where eccentricity can be thought of as a characteristic of a particular face (some faces are more oval than the others) while the circumference models the distance of the face from the capturing device. The bounding rectangle of the ellipse defines the actual slice to be decoded. These consideration lead to the definitions of respective scenario domains. We exemplify with scenario I^p_{ROI} domain definition (within a picture of $w \times h$) shown in (97).

$$
\begin{aligned}
X_{I^p_{\text{ROI}}} = \big\{ & p = (2 \cdot \xi_M \cdot 2 \cdot \xi_m)/(16 \cdot 16), p \in [1, P], && \text{(97a)}\\
& q \in [1, 16] && \text{(97b)}\\
& p' \geq \mu \cdot P, p + p' \leq P && \text{(97c)}\\
& o^2 = 4 \cdot \pi^2 (\xi_M^2 + \xi_m^2), o \geq O && \text{(97d)}\\
& \epsilon^2 \cdot \xi_M^2 = \xi_M^2 - \xi_m^2, \epsilon = E, 2 \cdot \xi_M \leq w, 2 \cdot \xi_m \leq h && \text{(97e)}\\
& a = a_{\text{ref}}, b = b_{\text{ref}}, c = c_{\text{ref}}, d = d_{\text{ref}}, && \text{(97f)}\\
& e = e_{\text{ref}}, f = f_{\text{ref}}, g = q_{\text{ref}}, h = h_{\text{ref}} \big\} && \text{(97g)}
\end{aligned}
$$

The number of macroblocks p within the slice is given by the area of the ellipse's bounding rectangle (cf. (97a)). Note that the size of a macroblock is 16×16 pixels. Depending on resolution, the picture/frame consists of maximally P macroblocks (cf. (97a)). The number of blocks within a macroblock q is constrained by (97b). It is known that o is always greater than a certain predefined constant O (cf. (97d)), i.e. O defines the maximal distance from the face to the camera. Furthermore, ϵ is equal to a constant E and the ellipse is entirely contained inside the picture/frame (cf. (97e)). Within a picture, it is assumed that the background always occupies the portion μ of the picture/frame comprising p' macroblocks (cf. (97c)). Referent actor execution times (cf. (97f) and (97g)) were taken from [4] and are expressed in cycles of the STMicroelectronics STxP70 processor.

Slices are sequenced as follows. First, I slices of both ROIs are decoded. This corresponds to the decoding of a complete I picture/frame. Thereafter, a number of i and Ii slices forming i and Ii picture/frames are decoded. This is first done for ROI and thereafter for the background. In reality, the number of i frames following I and Ii frames is bounded by the Group of Pictures length. For simplicity, we approximate this conservatively by allowing an arbitrary long sequence of i slices that is always followed by one Ii slice for both ROIs. Finally, the FSM revisits the initial state.

From the case study we see the two-level modeling flexibility our parameterized dataflow scenario concept offers. At the bottom level, within parameterized scenarios, it allows to express data dependent behavior using parameters, i.e. the behavior of a scenario is defined by the values parameters attain at run-time. These values depend on the characteristics of the input data (the input signal). In the case study, this is the relative displacement of the tracked object (face) and the camera and the ovality of the face. At the top level, the enclosing FSM is used to specify intricate control logic. In the case study, the control concerns the ordering of different types of slices.

In the exercise, we assume SDTV input format with signal type 480i 16:9 and resolution 720x480 pixels. Thus, $w = 720$, $h = 240$ and $P = 1620$. Furthermore, $O = 700$, $E = 0.6$ and $\mu = 30$. For these values using our performance analysis technique presented in Section 7 we obtain a conservative throughput estimate of $1.44252 \cdot 10^{-7}$ slices per cycle. If we were to use the FSM-SADF techniques using the upper endpoints of default parameter intervals we obtain a throughput estimate of $1.78516 \cdot 10^{-7}$. The comparison shows that our results tightens the FSM-SADF result by 19.19% due to the fact that FSM cannot be used to express complex data-dependent dynamics of a scenario.

9 Discussion and Conclusion

In this article, we have presented a novel dataflow formalism that combines FSM with parameterized dataflow as the underlying concurrency model. An FSM has the natural capability of expressing intricate control logic governing the application behavior, while parameterized dataflow is very fit in expressing fine-grained data-dependent application dynamics. The domain concept entails a modeling flexibility which allows to express analytical relationships between input data (signal) characteristics and their graph manifestations, i.e. graph rates and actor firing delays.

The model adopts the scenario-based modeling abstraction where the execution of an application is interpreted as a sequence of scenarios each modeled by a parameterized dataflow structure with the attached domain that captures the complex relationships between input signal and graph parameters. The scenario occurrence patterns are given by the scenario FSM. For a subclass of parameterized graphs where SDF serves as the base model, we developed novel techniques for worst-case performance analysis that by working directly with graph parameters avoid the need for enumeration of the respective domains.

As future work we plan to fully automate our technique (the exploration Algorithm 1 is performed manually) and perform detailed scalability analysis. We believe that our technique except in the tightness of generated worst-case performance estimates can outperform the existing techniques of FSM-SADF when analysing graphs with large repetition vectors. This is because simulation of graphs with large repetition vectors incurs a high penalty in time. Furthermore, we want to remove the restrictions concerning the structure of the input SDF-PDF specifications by performing a compositional analysis. In the analysis, "problematic" graph substructures would be abstracted into corresponding two-actor latency-rate structures. Furthermore, using the latency-rate abstraction as a key part of the compositional analysis approach we aim to explore worst-case performance analysis of parameterized scenario graphs with a less restrictive parameter change pattern, i.e. where parameters can change even within the graph iteration.

References

[1] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. Synchronization and linearity: an algebra for discrete event systems, 2001.

[2] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[3] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–10, Sept 2013.

[4] Vagelis Bebelis, Pascal Fradet, and Alain Girault. A framework to schedule parametric dataflow applications on many-core platforms. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 125–134, New York, NY, USA, 2014. ACM.

[5] B. Bhattacharya and S.S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *IEEE International Workshop on Rapid System Prototyping*, pages 84–89. IEEE Computer Society, 2000.

[6] B. Bhattacharya and S.S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *Signal Processing, IEEE Transactions on*, 49(10):2408–2421, Oct 2001.

[7] Shuvra S. Bhattacharyya, Ed F. Deprettere, and Bart D. Theelen. Dynamic dataflow graphs. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 905–944. Springer New York, 2013.

[8] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, Feb 1996.

[9] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in el greco. In *Hardware/Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on*, pages 142–146, May 2000.

[10] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432 vol.1, April 1993.

[11] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI signal processing systems for signal, image and video technology*, 19(2):179–194, 1998.

[12] Guy Cohen, Stphane Gaubert, and Jean-Pierre Quadrat. Max-plus algebra and system theory: Where we are and where to go now. *Annual Reviews in Control*, 23:207 – 219, 1999.

[13] Nathalie Cossement, Rudy Lauwereins, and Francky Catthoor. DF*: An extension of synchronous dataflow with data dependency and non-determinism. In *Forum on Design Languages, 2000. FDL 2000.*, 2000.

[14] K. Desnos, M. Pelcat, J.-F. Nezan, S.S. Bhattacharyya, and S. Aridhi. PiMM: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 41–48, July 2013.

[15] Pascal Fradet, Alain Girault, and Peter Poplavko. SPDF: A schedulable parametric data-flow moc. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 769–774, San Jose, CA, USA, 2012. EDA Consortium.

[16] S. Gaubert. Performance evaluation of (max,+) automata. *Automatic Control, IEEE Transactions on*, 40(12):2014–2025, Dec 1995.

[17] Stéphane Gaubert, Peter Butkovic, and Raymond Cuninghame-Green. Minimal (max,+) realization of convex sequences. *SIAM Journal on Control and Optimization*, 36(1):137–147, 1998.

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

[19] Marc Geilen and Sander Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 125–134, New York, NY, USA, 2010. ACM.

[20] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput analysis of synchronous data flow graphs. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, ACSD '06, pages 25–36, Washington, DC, USA, 2006. IEEE Computer Society.

[21] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, A.J.M. Moonen, M.J.G. Bekooij, B.D. Theelen, and M.R. Mousavi. Throughput analysis of synchronous data flow graphs. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 25–36, June 2006.

[22] A. Girault, Bilung Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(6):742–760, Jun 1999.

[23] D Grois and O Hadar. Recent advances in region-of-interest coding. *Recent Advances on Video Coding*, pages 49–76, 2011.

[24] Soonhoi Ha and Hyunok Oh. Decidable dataflow models for signal processing: Synchronous dataflow and its extensions. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 1083–1109. Springer New York, 2013.

[25] Elena Hammari, Francky Catthoor, Per Gunnar Kjeldsberg, Jos Huisken, K Tsakalis, and L Iasemidis. Identifying data-dependent system scenarios in a dynamic embedded system. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[26] Bernd Heidergott, Geert Jan Olsder, and Jacob Van Der Woude. *Max Plus at Work: Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications*. Princeton University Press, 2006.

[27] K.M. Kavi, B.P. Buckles, and U. Narayan Bhat. A formal definition of data flow graph models. *Computers, IEEE Transactions on*, C-35(11):940–948, Nov 1986.

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

[29] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *In Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 3–0531, 2003.

[30] M. Pankert, O. Mauss, S. Ritz, and H. Meyr. Dynamic data flow and control flow in high level DSP code synthesis. In *Acoustics, Speech, and Signal Processing, 1994. ICASSP-94., 1994 IEEE International Conference on*, volume ii, pages II/449–II/452 vol.2, Apr 1994.

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

[32] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[33] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr. High-level software synthesis for the design of communication systems. *Selected Areas in Communications, IEEE Journal on*, 11(3):348–358, Apr 1993.

[34] Hanif D Sherali and Warren P Adams. *A reformulation-linearization technique for solving discrete and continuous nonconvex problems*, volume 31. Springer Science & Business Media, 2013.

[35] F. Siyoum, M. Geilen, J. Eker, C. von Platen, and H. Corporaal. Automated extraction of scenario sequences from disciplined dataflow networks. In *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*, pages 47–56, Oct 2013.

[36] Firew Siyoum, Marc Geilen, Orlando Moreira, and Henk Corporaal. Worst-case throughput analysis of real-time dynamic streaming applications. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 463–472, New York, NY, USA, 2012. ACM.

[37] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411, July 2011.

[38] B. D. Theelen, M. C. W. 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 *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, MEMOCODE '06, pages 185–194, Washington, DC, USA, 2006. IEEE Computer Society.

[39] L. Thiele, K. Strehl, D. Ziegengein, R. Ernst, and J. Teich. Funstate-an internal design representation for codesign. In *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, pages 558–565, Nov 1999.

[40] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proceedingsof the 10th International Workshop on Software &Amp; Compilers for Embedded Systems*, SCOPES '07, pages 11–22, New York, NY, USA, 2007. ACM.

[41] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. Buffer capacity computation for throughput-constrained modal task graphs. *ACM Trans. Embed. Comput. Syst.*, 10(2):17:1–17:59, January 2011.

[42] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 183–194, April 2008.

[43] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.