Chapter 8: Scenarios in Dataflow Modelling and Analysis

Marc Geilen, Mladen Skelin, Reinier van Kampenhout, Hadi Alizadeh Ara,
Twan Basten, Sander Stuijk, Kees Goossens

Abstract Dataflow models can be used to model and program concurrent systems and applications. Static timed dataflow models commonly abstract the temporal behavior of systems in terms of their worst-case behaviors. This may lead to models that are very pessimistic. The scenario methodology can be applied to the dataflow modelling approach to group similar dynamic behaviors into static dataflow behaviors that abstract the system scenarios in a tight fashion. Constraints on the possible scenario transitions in the system can be modelled, among other options, by a finite state automaton. This approach leads to a model called Scenario-Aware Dataflow (SADF) that is presented in this chapter. We introduce the model and its semantics and discuss its fundamental analyis techniques. We discuss a parametrized extension and its analysis. We discuss a dataflow programming model and its implementation challenges. We give an overview of refined analysis techniques and run-time exploitation possibilities of SADF.

M.C.W. Geilen
Eindhoven University of Technology, e-mail: m.c.w.geilen@tue.nl
M. Skelin
Eindhoven University of Technology, e-mail: m.skelin@tue.nl
J.R. van Kampenhout
Eindhoven University of Technology, e-mail: j.r.v.kampenhout@tue.nl
H. Alizadeh Ara
Eindhoven University of Technology, e-mail: s.h.seyyed.alizadeh@tue.nl
T. Basten
Eindhoven University of Technology & ESI, TNO, e-mail: a.a.basten@tue.nl
S. Stuijk
Eindhoven University of Technology, e-mail: s.stuijk@tue.nl
K.G.W. Goossens
Eindhoven University of Technology, e-mail: k.g.w.goossens@tue.nl

1 Introduction

In this chapter we illustrate the application of the scenario methodology to the timed dataflow model of computation. Dataflow is an abstract mathematical model that can be used to model streaming applications and the resources on which they are realized. They are used, for instance, in the form od CSDF graphs in Chapter 4 to model applications on DVFS enabled processor platforms. Moreover, they can be effectively used to model flexible manufacturing systems, as discussed in detail in Chapter 9.

Figure 1 shows a dataflow model of an H.263 decoder. The circles are *actors*. They represent activities in the decoder. For example, the actor labelled vld represents the activity of the variable-length decoder. Actors execute their activities repeatedly. Actors have *dependencies* that determine their earliest activation times. An activation of an actor is called a *firing*. The dependencies are shown as incoming arrows, called *edges*, with *tokens* on them. There can be dependencies on firings of other actors, but also on earlier firings of the same actor or on external events. When an actor fires, it takes (consumes) tokens from its incoming dependencies. The firing takes a certain amount of time, after which the firing completes and produces tokens on the outgoing dependencies, thus satisfying the firing dependencies of other actors. It can also output tokens to the environment. Tokens in the model represent dependencies between firings. In the concrete systems such dependencies could be due to data dependencies between computations, but also due to resource dependencies, for example, the order of scheduling on a processor.

Different complementary views exist on dataflow models. Sometimes the functionality of actors is emphasized, where the model describes how actors compute output tokens with *values* from its input tokens with values, for example in Kahn Process Networks [24]. Usually, actors represent functional, deterministic and stateless behavior in terms of their computations. Another view on dataflow models emphasizes their timing and performance. This is the view we focus on in this chapter. In this view tokens do not carry values, but are pure dependencies. Actors have execution times that make that tokens are produced, or actors start, at certain moments in time. The timing has implications for performance properties such as throughput and latency. The models and their semantics are discussed in more detail in Section 3.

Dataflow models have a number of important strengths. They make *concurrency* very explicit. Actor firings that have no dependencies on one another are concurrent. The explicit concurrency can be exploited to optimize performance and for scheduling purposes. Many dataflow models (though not all) are *determinate*. In a determinate model, even though certain computations can be performed in arbitrary orders because of concurrency, the final result is independent of that order, which separates the concerns of correct functionality and scheduling. An important strength for the timed view on dataflow models is that they are *monotone*: if some input dependency is delayed (pushed into the future), then any event in the model (actor firing, token production) cannot happen earlier than its original time. Consequently, also, when an actor firing duration is shortened, no event in the model can occur later than in the

original model. This property allows one to make simple, deterministic worst-case abstractions of systems, which can be analyzed very efficiently. This is discussed in more detail in Section 2.

Dataflow models also have some weaknesses that need consideration. The more expressive dataflow models are generally also more difficult to analyse. Kahn Process Networks (KPNs) [24], for example, allow general determinate data-dependent behavior to be expressed, such as an actor that produces a different number of output tokens depending on the value of an input token. Analysis of KPN models, however, is very hard. The problem of finding minimal buffer sizes that allow deadlock-free execution, for example, is undecidable [34, 14]. At the other end of the spectrum we find dataflow models with limited expressiveness, such as Synchronous Data Flow (SDF) [27] in which the actors are deterministic and produce and consume fixed numbers of tokens. They are much more amenable to analysis and synthesis techniques, such as optimal scheduling [42] or buffer sizing [44, 31], although these problems are still in higher complexity classes [31]. These more static dataflow models do not suffice to model modern dynamic applications without being overly pessimistic about their performance and resource usage. Because of this trade-off between expressiveness and analyzability, many slightly different dataflow models have appeared in literature [46]. In this chapter we show how the scenario methodology can be applied to dataflow models to arrive at a model that strikes a particularly useful balance between the expressiveness needed to address the dynamic variation in modern applications and architectures, to limit the overestimation of resources and to preserve significant analysis and synthesis possibilities.

Dataflow models are suitable for applications that consist of fragments of sequential behavior that are internally deterministic and static and that are composed into parallel applications that may exhibit non-deterministic variation in execution of those behaviors. The dataflow models is also mostly suitable for applications that repetitively perform the same, or similar, behavior, possibly operating on or transforming streams of data.

The H.263 application is an example of an application that exhibits significant amounts of variation. In particular, the amount of data spent to encode the difference between successive video frames strongly depends on the degree of difference between the frames. This is expressed by the number of macro-blocks used to encode the difference. In the model, this is the number of tokens exchanged between the vld and the idct actors, indicated in Figure 1 with the consumption rate of n, tokens per firing, which may take different values for different frames. One can prove that a static model with a fixed number of blocks (the maximum value that n may take) is a conservative abstraction, but it is in many circumstances too pessimistic. Alternatively, a model can be made that exactly defines the number of macro blocks and how it depends on the data input, but such a model would be too complex for efficient analysis. The scenario methodology proposes to group the run-time situations that occur in a limited number of scenarios in which the worst-case resource usage overestimates the actual usage only by a moderate amount. We do this by grouping the number n of macro-blocks into ranges within which we use the maximum as a worst-case representative. For example, the run-time situations in which

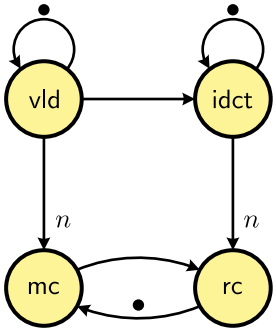

Fig. 1 A dataflow model of an H.263 decoder

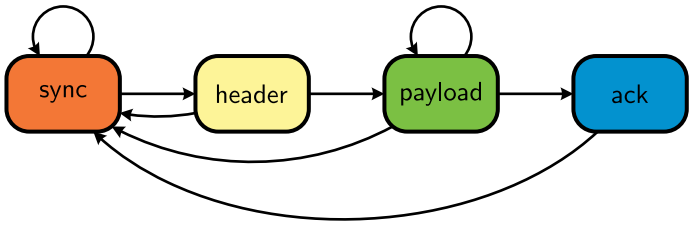

Fig. 2 Specification of the possible scenario sequences

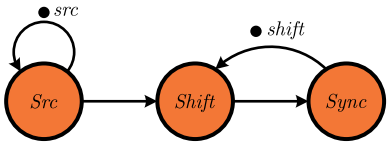

Fig. 3 Dataflow graph of the sync scenario

there are 30–40 macro-blocks may be represented by a single scenario in which the work load is assumed to corresponds to 40 macro-blocks, the worst-case of all run-situations covered by the scenario. Similarly, the actor execution times in the model are selected to correspond to worst-case execution times of the software. Besides the scenarios due to varying number of macro-blocks, in H.263 variation occurs due to the fact that some frames are encoded independently from previous frames. We capture the behavior of that run-time behavior by a separate scenario as well.

There are many examples where the same approach applies, for example in the channel equalizer model in [29]. In this system periodically one of every eight symbols triggers a channel estimation computation with extra computational requirements. In this case the dynamic variations exhibit deterministic periodic patterns. Cyclo-Static Dataflow (CSDF) models can be used to represent this behavior.

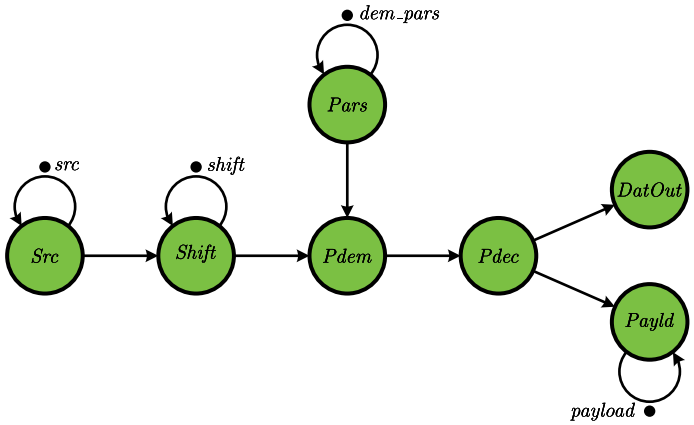

Fig. 4 Dataflow graph of the payload scenario

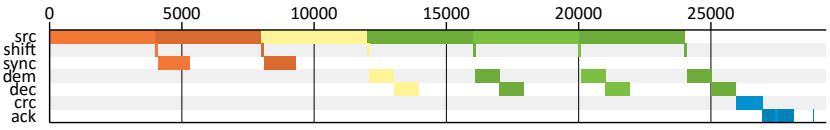

Fig. 5 Gantt chart of an execution of the WLAN model

The Scenario-Aware Dataflow (SADF) model, discussed in this chapter, generalizes CSDF.

A last example to mention is a WLAN receiver [30]. Figure 2 shows the finite state automaton that specifies the possible sequences of scenarios of this model. The reception of a frame consists of a sequence of different activities: synchronization, header processing, payload processing, ans positive or negative acknowledgement after error-detection. We use scenarios to capture the variations in the run-time situations associated with these activities. Switches between activities (scenarios) are non-deterministic. Synchronization with the sender may be lost at any time and a frame may contain an different numbers of payload symbols. They can therefore not be expressed by CSDF. However, their occurrences are constrained to particular patterns. We want to be able to exploit this knowledge. The finite state automaton of Figure 2 encodes the possible sequences of scenarios. Loss of synchronization causes a transition back to the synchronization state to be taken. The self-transition on the payload state is used when another payload symbol follows in the frame. When the frame ends, the scenario changes to the acknowledgement scenario. Note that it is conservative in the sense that it still allows some sequences of scenarios that cannot occur in reality. For instance, it allows for arbitrarily long sequences of payload symbols, even though under the real protocol constraints there can be no more than 256. This could be encoded in a finite state automaton, but it would have a large number of states. Possibly more compact representations could be applied

such as the regular expressions used by Ara et al. [1] to obtain a compact and exact representation.

The behavior in the individual scenarios is defined by dataflow graphs. Figure 3 shows the behavior of the sync scenario and Figure 4 shows the behavior of the payload scenario. Other examples of applications modelled with SADF can be found in literature [1, 29, 32, 47].

2 Scenarios in Dataflow Modelling

2.1 Relation to the Scenario Methodology

The essential property of the SADF model is that it maintains as much as possible of the determinacy of dataflow behavior, while introducing the possibility for non-deterministic variation in the form of scenarios as elaborated on the H.263 decoder of Figure 1. Every scenario is represented by an SDF graph that represents the worst-case from a multidimensional cost-perspective within the cluster of run-time situations it represents.

The concept of SADF lies perfectly within the general scenario methodology as discussed in Chapter 2. In particular, the system-scenario methodology is a five step approach in which each step has a design-time and a run-time phase. The first step of the methodology is *identification*, performed at design-time, in which identified run-time situations are clustered into scenarios based on a particular multidimensional cost perspective. In terms of SADF, the identification step entails the identification of SDF-like regions in dynamic systems under consideration. The so-identified static regions are called scenarios. An aspect that is particular to the dataflow approach is that such regions can span activities in both *time* and *space*. For example, in the Gantt chart of the WLAN example in Figure 5 we observe that the scenarios (differently colored parts) follow a pipelined structure and overlap in time. More about methodologies that enable the identifications of these regions can be found in [37, 20].

The second step of the system scenario-methodology is known as *prediction*, where a scenario has to be selected from a scenario set based on the parameters of the run-time situation. In SADF the selection options can be constrained by a model that defines the possible scenario occurrence patterns [47, 46]. In case a finite state automaton os used, the automaton is generally non-deterministic where the non-determinism may have one of the two following roles: a *descriptive* and a *constraining* role [23]. Within the descriptive role, non-determinism captures an aspect of the world that is not completely known and that behaves in an unpredictable manner. Within the constraining purpose, the non-determinism designates different possibilities for implementation. Within the system-scenario methodology the descriptive non-determinism is more pronounced in the sense that it arises from the deliberated decision to ignore the facts which influence the selection [22]. In soft-

ware, these facts are often values of certain control variables. In a cyber-physical system, they could be uncontrollable events. Section 4 discusses how the, in the former situation, run-time prediction activities can be made explicit if SADF is used as a programming model. In addition to non-determinism, SADF supports the specification of probability mechanisms that govern the choice between scenarios [47]. Such quantitative information about the likelihood of scenario transitions can be used for analysis purposes or to decide on scenario switching activities when cost are involved with the switching itself.

The third step is called *exploitation*. At-design time, it involves optimization that is applied based on the knowledge of the scenario structure and patterns and at run-time it is the execution of the scenario. In the SADF context, the design-time phase involves SDF-related transformations like multi-rate expansion, retiming, pipelining and unfolding with the purpose of optimizing graph throughput, minimizing code size, mapping to processors and memory, etc. [3, 42]. The run-time phase entails scheduling and execution of the scenario SDF graph.

The fourth step of interest is called *switching*, which is the act of run-time reconfiguration to enable the execution of another scenario. In the context of SADF and software-intensive systems, each scenario in the SADF graph could, for example, use a different mapping to processors or memory. To implement this, a run-time reconfiguration mechanism is employed that can transfer data items (tokens) and code (actors) between different memories whenever a scenario switch occurs [46]. Stuijk et al. [45] present a design flow that maps a throughput-constrained application, modeled with an SADF to an MPSoC.

The final, fifth step is called *calibration* that collects information about values of run-time parameters and further adjusts the system to optimize against a certain cost function. Calibration is only meaningful when performance constraints are soft constraints or to optimize average-case behavior in the absence of constraints. In the context of SADF, this step may have counterparts, such as calibration of worst-case executions times, but such approaches have not been worked out as yet.

2.2 Abstraction and Refinement in Timed Dataflow

We represent run-time situations in a scenario by worst-case behavior of the scenario, or a tight upper bound on its worst-case behavior. In general systems it may be difficult to identify the worst-case behavior due to complex interactions between components or the environment, or due to resource arbitration. The longest execution time of a single task may, for instance, not always lead to the longest execution time of the overall application. Such an effect is often called a *timing anomaly* [28].

Dataflow models are monotone (more generally, max-plus linear, see Section 3.2) and do not exhibit such timing anomalies. This has the advantage that the worst-case situation can be easily identified and corresponds to the actors taking their largest execution times. This holds within a single dataflow graph, but also compositionally, i.e., when they are placed in a context, for example, when dataflow graphs are

built hierarchically in a modular fashion. If a dataflow graph consumes inputs from its environment and produces outputs to the environments, then it can be formally shown that any run-time behavior represented by the scenario performs at least as good as the scenario representative in the following sense. Each of the outputs are produced at times, no later than they are produced in the representative behavior if the inputs are provided no later than in the representative behavior. This establishes a very precisely defined, formal abstraction-refinement relation between the run-time situations in the scenario and the representative behavior [17], in which the representative behavior is the *abstraction* and the concrete run-time situations are *refinements* of that behavior.

An important property of the abstraction-refinement relation is that non-deterministic systems (executions may vary non-deterministically under data or resource dependencies) can have deterministic abstractions (a dataflow model with fixed, deterministic, worst-case execution times). This has significant advantages for performance analysis. Only a single, deterministic behavior (the abstraction) needs to be verified and if it satisfies its performance requirements then all the non-deterministic refinements are guaranteed to also satisfy those performance constraints. This avoids many of the state-space explosion issues associated with the verification of non-deterministic systems [36].

Within the scenario approach, the SADF model is used to define such an abstraction of the actual behavior and concrete run-time situations of the system. Moreover, the explicit goal is to define the scenarios such that the abstraction is not overly pessimistic compared to the run-time situations that it represents.

3 Modelling and Analysis of Scenario-Aware Dataflow

3.1 The Scenario-Aware Dataflow Model

This section gives a detailed definition of the Scenario-Aware Dataflow model. We present a formal definition in which we abstract from the streams of data values that are exchanged in the models as well as from the functions that the components compute. We make this abstraction, because we primarily deal with the timing, performance and resource usage of the models to act as models of scenarios, for which the data values are irrelevant. We do not deal with the functionality that the dataflow graphs realize. An SADF graph consists of a finite set S of scenarios, a mapping $\sigma : S \to \mathscr{G}$ that maps each scenario to a static dataflow graph, i.e., an SDF graph, for that scenario ($\mathscr{G}$ denotes the set of all static dataflow graphs), and a language $\mathscr{L} \subseteq S^\omega$ that defines all the possible, infinitely long, scenario sequences. We consider infinitely long scenarios sequences, because many applications we consider are stream processing applications or continuous production machines. The model and theory can, however, be similarly applied to finite scenario sequences. In the

WLAN example, $S = \{\mathsf{sync}, \mathsf{header}, \mathsf{payload}, \mathsf{ack}\}$. The definition of the language of scenario sequences for the WLAN application is discussed later.

A static dataflow graph $\sigma(s)$ of a scenario $s \in S$ consists of a set A of *actors* and a set $D \subseteq A \times A$ of *dependencies* between these actors. (For simplicity we assume that at most one dependency exists between any pair of actors, although such a restriction is not necessary, for instance by defining D as a multiset.) Actors have an *execution time*, given by the function $\tau : A \to \mathbb{R}^{\geq 0}$. A dependency $(a_1, a_2) \in D$ expresses that the firings of actor a_2 depend on the firings of actor a_1. The graph associates with a function $i : D \to \mathbb{N}$ to every dependency $d \in D$ a number, $i(d)$, that determines the precise dependency of firings as follows. In a so-called single-rate graph, it enforces a dependency of firing $n + i(a_1, a_2)$ of actor a_2 on the completion of firing n of actor a_1 for all $n \in \mathbb{N}$. Operationally, we can think of actor a_1 *producing* tokens (one for each firing in a single-rate graph) that are subsequently *consumed* by firings of actor a_2 (also one for each firing in single-rate graph), with $i(a_1, a_2)$ tokens being initially present. Not all graphs are single-rate graphs, i.e., produce and consume exactly one token with each firing and each dependency. Graphs that are not single-rate are called *multi-rate* and some of their actors produce or consume larger, but fixed, quantities of tokens with each firing. For example, the n tokens consumed by a firing of actor mc from the actor vld in the H.263 model of Figure 1. We use $\pi(a_1, a_2)$ to denote the rate at which actor a_1 produces tokens on the dependency (a_1, a_2) and $\psi(a_1, a_2)$ to denote the rate at which actor a_2 consumes tokens. In general, firing k of a_2 depends on firing m of a_1 if the ranges $[k \cdot \psi(a_1, a_2), (k+1) \cdot \psi(a_1, a_2) - 1]$ and $[m \cdot \pi(a_1, a_2) + i(a_1, a_2), (m+1) \cdot \pi(a_1, a_2) + i(a_1, a_2) - 1]$ overlap, i.e., if firing m of a_1 produces a tokens that is consumed by firing k of a_2.

Note that firings of an actor have a logical ordering (when we talk about firing k of some actor a). Most of the time, the logical ordering coincides with the temporal ordering in which the firings occur, but this is not necessarily the case. Firings of the same actor can be concurrent (this is called *auto-concurrency*) and occasionally even out of logical order (some firing k starts before firing m although $k > m$). In models in which firings necessarily occur in the same order, logically and temporally, the token dependencies exchanged between actors can be seen as a FIFO queue and are also often implemented in that way. If firings can be out of order then a more general implementation is needed, such as a windowed cyclic buffer [4]. Many discussions in literature assume that firings occur in temporal order and that dependencies are FIFOs, but such a restriction is not necessary. When actors can complete their firings out of order and the dependencies are realized in FIFO order then the functionality of the graph may be compromised.

In a single scenario instance, the actors in the dataflow graph of that scenario fire a fixed number of times. Often, this is a minimal, non-empty collection of actor firings after which the graph returns to its original state in terms of tokens and dependencies. This collection of actor firings is called the *repetition vector* of the graph [27, 42]. This kind of scenario is called a *strongly consistent* scenario [47]. In general, however, an arbitrary collection of firings may be defined, denoted with a function $\rho : A \to \mathbb{N}$ that assigns the number of firings to each actor. This may leave the tokens on different edges in the graph than where they were at the start of the

scenario. An SADF graph that uses this is called a *weakly consistent* graph [15], assuming that it is still consistent in the sense that in the long run, no matter which scenario sequence it executes, the graph does not deadlock and the number of tokens that can accumulate on any channel is bounded a priori. In general, we will associate a scenario with an SDF graph and a corresponding repetition vector.

For performance analysis purposes, a *reward* specification $r : S \to \mathbb{R}$ may be additionally defined that associates with every scenario a real-valued quantity that captures the amount of progress that is made by that scenario. For example, the throughput requirement for the WLAN application is to process one OFDM symbol every 4μs, but only the scenarios sync, header and payload process one OFDM symbol. The ack scenario does not. This is captured by assigning a reward of 1 to the former scenarios and a reward of 0 to the latter.

Finally, an SADF model needs to define dependency relations across scenarios, for instance, in the WLAN model the processing of a symbol updates the channel estimation and synchronization parameters. Those results are inputs to the processing of the following scenario. Tokens are used in the SADF model to represent such dependencies. This is done by defining for every scenario a set of *initial tokens*, tokens that are present in the graph in its initial state, that carry dependencies from earlier scenarios, and *final tokens*, tokens left in the graph at the completion of the scenario, that can carry over dependencies to the following scenarios.

3.2 The Semantics of Scenario-Aware Dataflow

The behavior of an SADF graph is non-deterministic in terms of the sequence of scenarios from the language $\mathscr{L}$ that it executes. This is in fact the only non-deterministic element in the model, the scenario behaviors themselves are deterministic. This non-deterministic behavior of the model can be due to, for instance, data-dependent behavior in the actual system that the model abstracts from. For instance the number of macro-blocks in a video frame in the H.263 decoder is encoded in the video bit stream, but the model abstracts from the values in that stream. This abstraction is discussed in more detail in Section 4.

Figure 5 shows the behavior of the WLAN receiver that corresponds to a scenario sequence starting with $\mathsf{sync} \cdot \mathsf{sync} \cdot \mathsf{header} \cdot \mathsf{payload} \cdot \mathsf{payload} \cdot \mathsf{payload} \cdot \mathsf{ack}$ in the form of a Gantt chart. Different colors have been used in the figure to represent different scenarios. The rows in the chart correspond to actors and show the firings of those actors. Actors start their firings as soon as all dependencies are satisfied and the firings take an amount of time that is fixed per scenario, but may differ between scenarios. Some actors may fire only in some scenarios. In the example, actors crc and ack fire only in the ack scenario (colored blue).

An important observation to make is that the executions of the scenarios overlap in time. This happens because the different dependencies that carry over from one scenario to the next are satisfied at different points in time and the actor firings in the upcoming scenario start as soon as possible. This is an important advantage of the

SADF model, because despite this pipelined execution, it can deal with scenarios in isolation in a compositional manner for all analysis and synthesis purposes.

The view of the behavior of an SADF presented above is called its *operational semantics* and it is the most intuitive view to understand how the model operates. However, it is not the most convenient for mathematical analysis. To understand the mathematical properties of the model, from a temporal point of view, we observe that actor firings wait until all of their dependencies are satisfied, after which they fire for a constant duration. The completion of the firing, in turn, satisfies new dependencies. The time of enabling of the firing can be computed as the maximum of the times at which the individual dependencies are satisfied and the completion time of the firing is computed by adding the execution time to that. We see that the equations that determine how fast the graph executes are constructed from the mathematical operators max and $+$. A lot is known about the algebra that emerges from these two operators [2, 21], which is called max-plus algebra. In particular, and very importantly, it is known to be a linear algebra and it enjoys many of the properties of common linear algebra and has been extensively studied in literature [2, 21, 7, 11].

One of the properties of linear algebra that we use, is the fact that a linear system has a canonical representation as a matrix that computes the output and/or next state from the inputs and/or starting state. In the case of a static dataflow graph we consider the initial *state* of the graph as the time stamps (sometimes called *daters*) of the initial tokens in the graph, the times at which the initial dependencies are satisfied. Some graphs have open inputs consuming tokens from the environment. In that case the time stamps of those tokens (not their values!) are considered the inputs of the linear system. After completion of the collection of firings we model by the matrix, the time stamps of the final tokens in the graph represent the next state and any tokens produced on open outputs are the outputs of the system. This leads to an equation of the following form [17, 39]:

$$\begin{bmatrix} \mathbf{x}[k+1] \\ \mathbf{y}[k] \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}[k] \\ \mathbf{u}[k] \end{bmatrix} \tag{1}$$

In this equation, the vector $\mathbf{u}$ represents the inputs, vector $\mathbf{x}[k]$ the current state, vector $\mathbf{y}$ the outputs and vector $\mathbf{x}[k+1]$ the next state. $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are appropriately chosen matrices that precisely characterize the temporal behavior of the scenario. Note that the matrix-vector multiplication in this equation is in max-plus algebra, not classical linear algebra.

In case the model is closed, i.e., if it does not have inputs or outputs, then the following simple equation remains:

$$\mathbf{x}[k+1] = \mathbf{A}\mathbf{x}[k]$$

In an SADF graph, every scenario s can be individually characterized by a set of matrices $\mathbf{A}_s$, $\mathbf{B}_s$, $\mathbf{C}_s$ and $\mathbf{D}_s$. As an execution follows a particular scenario sequence from the language $\mathscr{L}$, the behavior is determined by a sequence of multiplications with matrices corresponding to the scenarios. In linear systems terminology, an SADF graph is a *switched linear system* in the max-plus linear algebra [48].

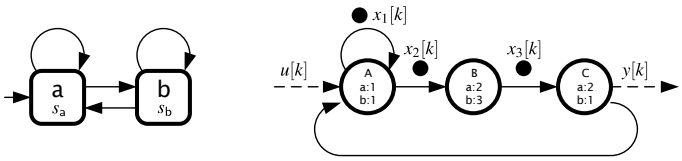

Fig. 6 Example SADF graph.

We exemplify the max-plus representation using the example SADF of Figure 6. The scenarios are shown in the rightmost part of the figure. We initially assume that the dashed arrows, representing open input and output channels, are not present. The graph has two scenarios: a and b. Each scenario graph consists of three actors: A, B and C. Operationally, a scenario consists of a single firing of each of the actors. The difference between the scenarios is only in the firing delays of actors B and C. In particular, in scenario a the firing of actors B and C will take 2 time units each, while in scenario b the firings will take 3 and 1 time units, respectively. Consequently, the scenario matrices differ. The matrices can be computed as explained in [16]. Here, we present the outcomes:

$$\mathbf{A}_\mathsf{a} = \begin{bmatrix} 1 & -\infty & 3 \\ 1 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix} \quad \text{and} \quad \mathbf{A}_\mathsf{b} = \begin{bmatrix} 1 & -\infty & 2 \\ 1 & -\infty & 2 \\ -\infty & 3 & -\infty \end{bmatrix}.$$

In the matrices, entry $[\mathbf{A}]_{i,j}$ specifies the time distance between the time stamp of initial token j and the time stamp of the final token i. In our example, the initial tokens coincide with the final tokens with token indices increasing from left to right w.r.t. the rightmost graph in Figure 6. An entry $-\infty$ indicates that the corresponding tokens are independent.

We now turn our attention to the dashed input and output channels of the scenario graphs of Figure 6. Their behavior is included by also specifying the matrices **B**, **C** and **D** of Equation 1. Matrix **B** captures the dependency of the final internal state on the input token. In this case it is a column vector as there is only one input token consumed. It is identical for both scenarios and has the value:

$$\mathbf{B}_\mathsf{a} = \mathbf{B}_\mathsf{b} = \begin{bmatrix} 1 \\ 1 \\ -\infty \end{bmatrix}$$

Only the left two tokens, x_1 and x_2, depend on the input and the timing distance is the execution time of actor A. Matrix **C** represents the dependency of the output token on the tokens of the initial internal state. In this case it is a row vector with the following values for the scenarios:

$$\mathbf{C}_\mathsf{a} = \begin{bmatrix} -\infty & -\infty & 2 \end{bmatrix} \quad \text{and} \quad \mathbf{C}_\mathsf{b} = \begin{bmatrix} -\infty & -\infty & 1 \end{bmatrix}$$

In both cases the output only depends on the rightmost token, x_3, and the timing distance is the execution time of actor C. The final part is the matrix $\mathbf{D}$, which in this case reduces to a scalar, because there is only one input and one output, with the trivial value of $-\infty$, because there is no direct dependency from the input to the output.

$$\mathbf{D}_{\mathsf{a}} = \mathbf{D}_{\mathsf{b}} = -\infty$$

The characteristic matrices completely define the timing of the scenarios. The behavior of the SADF consists of sequences of scenarios in the language $\mathscr{L}$. The language of scenario sequences can be specified with formalisms to define languages over a finite alphabet (in this case the set of scenarios). Well-known examples are Finite-State Automata (FSAs) and regular expressions (both are in fact equally expressive). In the example SADF of Figure 6, the scenario sequences are specified by the FSA on the left. It defines the language of all sequences of scenarios a and b that start with the scenario a. In [1], the language of scenario sequences is expressed with regular expressions including an explicit repetition construct, leading to a compact representation with an efficient analysis.

One can optionally include information about the likelihood of the occurrence of sequences of scenarios. For instance, by adding probabilities to the finite state machine, resulting in a Markov Chain representation that defines a σ-algebra on the language $\mathscr{L}$. This gives the model well-defined notions of stochastic behavior, such as *expected* throughput or *variance* in latency [47].

The semantics we have introduced can be used to define the explicit *state space* of an SADF graph in which the scenario sequences are defined by an FSA. The *state* of an SADF can be captured by a combination of the current state of the FSA and the current time-stamp vector of the tokens. For example, the initial state of the SADF graph in Figure 6 is the pair $\left(s_{\mathsf{a}}, \begin{bmatrix}0 & 0 & 0\end{bmatrix}^T\right)$. After executing the scenario a and the FSA non-deterministically moving to state s_{b}, the state would become $\left(s_{\mathsf{b}}, \begin{bmatrix}3 & 3 & 2\end{bmatrix}^T\right)$. Note that in many discrete as well as continuous and hybrid models, the state of a state space refers to a snapshot of the integral system at some point in the (physical or modelling) time domain. For SADF, and max-plus models in general, this is not the case. It refers, instead, to the state before or after the execution of certain scenarios, where the elements of the time-stamp vector of such a state refer to possibly different time stamps in the time domain.

Every scenario execution leads to a discrete transition in the state space as the FSA moves to a new state and the time-stamp vector changes accordingly. Naturally such a state space would be infinite, as the time stamps increase and diverge towards infinity. Such an infinite state space could not be constructed in practice and could not be used for any analysis. We therefore apply a normalization strategy to keep the state space finite. This strategy is based on the observation that the scale of a time-stamp vector has no impact on the possible future behaviors of the SADF in a given state. If an SADF, from a given state $(s,\mathbf{x})$, can perform a sequence of scenarios leading to a state $(s',\mathbf{x}')$, then the state $(s,\mathbf{x}+c)$, can perform the same sequence of scenarios leading to the state $(s',\mathbf{x}'+c)$ for any $c \in \mathbb{R}$. Therefore, we only explicitly record a state using its normalized time-stamp vector, i.e., as $\mathbf{x}-c$ for $c \in \mathbb{R}$ such

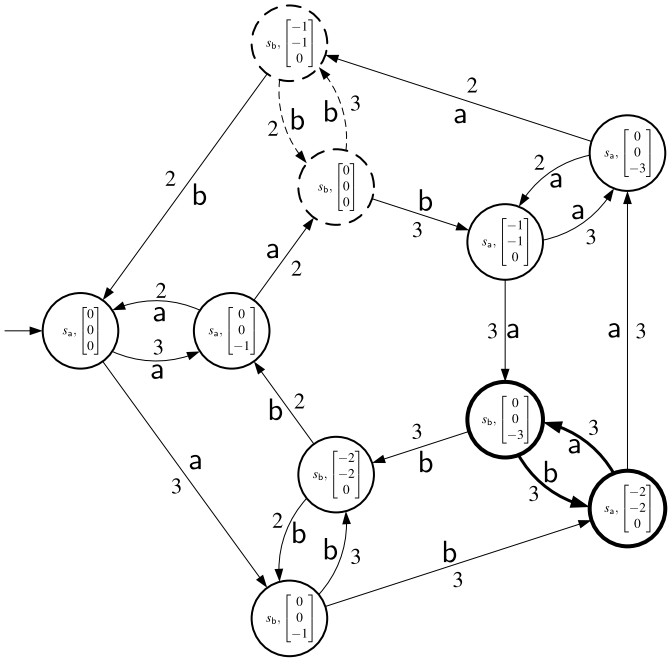

Fig. 7 State space of the SADF of Figure 6.

that $|\mathbf{x}-c|=0$. Hence, the state $\left(s_\mathrm{b}, \left[3\ 3\ 2\right]^T\right)$ is recorded as $\left(s_\mathrm{b}, \left[0\ 0\ -1\right]^T\right)$. This way the relative differences of the time stamps are recorded, but not their absolute values. To be able to account for the amount of time passing in the transition from the initial state to this state, the normalization constant c (in the example c is equal to 3) is associated with the state transition. If we follow this approach for the example of Figure 6, we arrive at the finite state space that is shown in Figure 7 (the bold and dashed arrows and circles are explained in the following section). The transitions in the state space are additionally decorated with the scenario that is executed.

The precise definition of the state space of an SADF is as follows. The set Σ of states of the state space are pairs:

$$\Sigma = \{(q,\mathbf{x}) \in Q \times (\mathbb{R} \cup \{-\infty\})^n \mid |\mathbf{x}| = 0\},$$

where Q is the set of states of the FSA and n is the size of the state vector. The transitions $((q_1,\mathbf{x}_1),d,(q_2,\mathbf{x}_2))$ of the state space are triples from $\Sigma \times \mathbb{R} \times \Sigma$, such that (i) there is a transition from state q_1 labelled with scenario s, to state q_2 in the FSA (ii) $\mathbf{x}_2 + d = \mathbf{A}_s\mathbf{x}_1$. We usually refer to the state space as only the set of all states that are reachable from the initial state $(q_0,\mathbf{0})$, where q_0 is the initial state of the FSA, and the corresponding transitions. Moreover, for convenience, we may additionally label the transitions in the state space with the scenarios that they

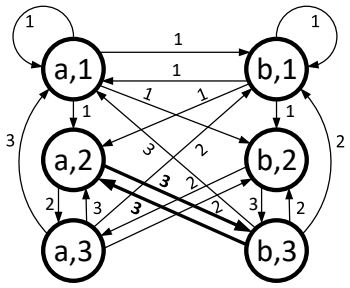

Fig. 8 Max-plus automaton graph of the SADF of Figure 6.

correspond to and/or with the rewards corresponding to these scenarios, as we have done in the state space in Figure 7.

The state space allows us to determine the state vector obtained after any finite scenario sequence that is a prefix of a word in the scenario language as follows. We follow a path labelled with the scenarios in the scenario sequence through the state space, starting from the initial state. If the sum of the edge weights on the path is d and the normalized time stamp vector of the final state is $\mathbf{x}$, then the final state-vector after the scenario sequence is equal to $\mathbf{x}+d$.

3.3 Performance Analysis of Scenario-Aware Dataflow

If the language of scenario sequences is a regular language, like in the example of Figure 6, and defined by an automaton without acceptance conditions, then the structure is called a *max-plus automaton* [12]. There are known techniques to compute from such a max-plus automaton the worst-case number of scenarios per time unit [12], or, if the scenarios are annotated with rewards, the worst case total reward per time unit [16, 15]. Those methods can also report the critical scenario sequence and the critical path of actor firings within the scenarios. It the automaton does have acceptance conditions, then the same analysis can be applied, also with exact results, by identifying and subsequently removing any states that can only occur a finite number of times in any accepted word, using standard automata analysis techniques.

The worst-case throughput analysis centers around the structure called Max-Plus Automaton Graph (MPAG). In particular, given an SADF graph, the structure is constructed as follows: a vertex is created for each initial token of a scenario of an FSA state. If $[\mathbf{A}_{\mathsf{s}}]_{i,j} \neq -\infty$ and there is a transition in the FSA from state m labeled with scenario r to a state n labeled with scenario s, an edge is created from node j in state m to node j in state n with weight $[\mathbf{A}_{\mathsf{s}}]_{i,j}$. The MPAG of the SADF of Figure 6, is shown in Figure 8. Maximum Cycle Mean (MCM) [10] analysis of the MPAG structure will identify the critical scenario sequence, i.e., the sequence with

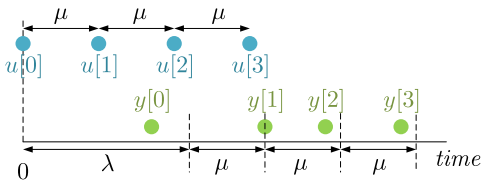

Fig. 9 Definition of latency

the worst-case average amount of time taken per scenario. The inverse of the MCM equals the worst-case throughput of the graph, the worst-case number of scenarios executed per time unit. For the running example of Figure 6, the critical scenario sequence is captured with bold arrows in Figure 8. This is the sequence $(\mathsf{ab})^{\omega}$. The corresponding MCM is equal to 3 defining the greatest lower bound on throughput of any scenario sequence, which is equal to $1/3$ scenarios per time unit. Note that in practice, a scenario often represents a coherent set of computations, like decoding one audio/video frame. As explained earlier, in some SADF models, the amount of progress differs per scenario, as in the WLAN example. In such cases rewards are used to specify the progress. The MPAG is then extended by annotating the edges additionally with the reward of the corresponding scenario and a Maximum Cycle Ratio (MCR) analysis is performed in place of the MCM analysis [15]. The MCR gives us the worst-case (minimum) amount of reward per time unit.

The linear model also facilitates the computation of latency. Latency is often defined with respect to a periodic input that delivers inputs to the system starting at time 0 and with some period $\mu > 0$. This is illustrated in Figure 9. We assume a single input $u[k]$ and a single output $y[k]$, but the definition and the analysis are easily generalized to multiple inputs and outputs. Latency is defined as the smallest value $\lambda \in \mathbb{R}$ such that $y[k] \leq \lambda + k \cdot \mu$ for all $k \in \mathbb{N}$. For the example of Figure 9, for any smaller value of λ than indicated, $y[1] > \lambda + 1 \cdot \mu$. If such a value of λ is found, then the system is a refinement, in the sense of Section 2.2, of a system that produces outputs periodically with period μ after an initial delay of λ.

Note that there may be a trade-off between throughput and latency. For a higher throughput (smaller value of the period μ, the latency may be larger. Their relation is always monotone. Moreover, if $\frac{1}{\mu}$ is higher than the maximal throughput of the graph, then such a value λ does not exist and the graph does not have a latency.

For multiple outputs, the definition is generalized as follows. The latency is the smallest vector $\boldsymbol{\lambda}$ such that

$$\mathbf{y}[k] \leq \boldsymbol{\lambda} + k\mu \quad \text{for all} \quad k \in \mathbb{N} \,.$$

This vector can be computed as:

$$\boldsymbol{\lambda} = \max_{k \in \mathbb{N}} \mathbf{y}[k] - k\mu \,.$$

In max-plus algebra it is common to use $\oplus$ as short-hand notation for the binary max operator and $\bigoplus$ for the max quantifier. In this notation we get the following equation:

$$\boldsymbol{\lambda} = \bigoplus_{k \in \mathbb{N}} \mathbf{y}[k] - k\mu \ ,$$

Following straightforward max-plus linear algebra computations we can derive that the latency for a static dataflow graph with one scenario with matrices **A**, **B**, **C**, and **D**, is computed as follows [1]:

$$\boldsymbol{\lambda} = \mathbf{C}\left(\mathbf{A} - \mu\right)^{*}\left(\mathbf{x}[0] \oplus \left(\mathbf{B0} - \mu\right)\right) \oplus \mathbf{D0} \ , \tag{2}$$

where the $*$-closure of a square matrix **M** is defined as

$$\mathbf{M}^{*} = \bigoplus_{k=0}^{\infty} \mathbf{M}^{k} \ .$$

Note that this closure exists in the latency computation of Equation 2 if and only if the graph has a latency. It can be efficiently computed exactly. We observe further that the latency also depends on the initial state, $\mathbf{x}[0]$ of the system.

We next show that the derivation of the latency of a max-plus linear system is a straightforward exercise. The following equation can be shown to hold for the state vector $\mathbf{x}[k]$ by induction on k and the fact that $\mathbf{x}[k+1] = \mathbf{A}\mathbf{x}[k] \oplus \mathbf{B}\mathbf{u}[k]$.

$$\mathbf{x}[k] = \mathbf{A}^{k}\mathbf{x}[0] \oplus \left(\bigoplus_{m=0}^{k-1} \mathbf{A}^{m}\mathbf{B}\mathbf{u}[k-m-1] \right)$$

Then the output vector can be computed as

$$\begin{aligned} \mathbf{y}[k] &= \mathbf{C}\mathbf{x}[k] \oplus \mathbf{D}\mathbf{u}[k] \\ &= \mathbf{C}\left(\mathbf{A}^{k}\mathbf{x}[0] \oplus \bigoplus_{m=0}^{k-1} \mathbf{A}^{m}\mathbf{B}\mathbf{u}[k-m-1] \right) \oplus \mathbf{D}\mathbf{u}[k] \ . \end{aligned}$$

From this the latency can be computed using the periodic inputs $\mathbf{u}[k] = k \cdot \mu \cdot \mathbf{0}$

$$
\begin{aligned}
\boldsymbol{\lambda} &= \bigoplus_{k\in\mathbb{N}} \mathbf{y}[k] - k\cdot\mu \\
&= \bigoplus_{k\in\mathbb{N}} \mathbf{C}\left(\mathbf{A}^k\mathbf{x}[0] \oplus \bigoplus_{m=0}^{k-1} \mathbf{A}^m\mathbf{B}\mathbf{u}[k-m-1]\right) \oplus \mathbf{D}\mathbf{u}[k] - k\cdot\mu \\
&= \mathbf{C}\bigoplus_{k\in\mathbb{N}} (\mathbf{A}-\mu\mathbf{I})^k\mathbf{x}[0] \oplus \bigoplus_{k\in\mathbb{N}}\left(\mathbf{C}\bigoplus_{m=0}^{k-1}(\mathbf{A}^m\mathbf{B}\cdot(k-m-1)\cdot\mu\cdot\mathbf{0}) \oplus \mathbf{D}\cdot k\cdot\mu\cdot\mathbf{0}\right) \\
&\qquad\qquad - k\cdot\mu \\
&= \mathbf{C}(\mathbf{A}-\mu)^*\mathbf{x}[0] \oplus \bigoplus_{k\in\mathbb{N}}\left(\mathbf{C}\bigoplus_{m=0}^{k-1}\mathbf{A}^m\mathbf{B}\cdot(k-m-1)\cdot\mu\cdot\mathbf{0} - k\mu\right) \oplus \bigoplus_{k\in\mathbb{N}}\mathbf{D}\cdot k\cdot\mu\cdot\mathbf{0} \\
&\qquad\qquad - k\cdot\mu \\
&= \mathbf{C}(\mathbf{A}-\mu)^*\mathbf{x}[0] \oplus \mathbf{C}\bigoplus_{k\in\mathbb{N}}\left(\bigoplus_{m=0}^{k-1}(\mathbf{A}-\mu)^m\right)\mathbf{B}\cdot(-1)\cdot\mu\cdot\mathbf{0} \oplus \mathbf{D}\mathbf{0} \\
&= \mathbf{C}(\mathbf{A}-\mu)^*\mathbf{x}[0] \oplus \mathbf{C}\left(\bigoplus_{k\in\mathbb{N}}\bigoplus_{m=0}^{k-1}(\mathbf{A}-\mu)^m\right)(\mathbf{B}\mathbf{0}-\mu) \oplus \mathbf{D}\mathbf{0} \\
&= \mathbf{C}(\mathbf{A}-\mu)^*\mathbf{x}[0] \oplus \mathbf{C}(\mathbf{A}-\mu^*)(\mathbf{B}\mathbf{0}-\mu) \oplus \mathbf{D}\mathbf{0} \\
&= \mathbf{C}(\mathbf{A}-\mu)^*(\mathbf{x}[0] \oplus (\mathbf{B}\mathbf{0}-\mu)) \oplus \mathbf{D}\mathbf{0}
\end{aligned}
$$

Note that although the definition is given for a periodic input, the definition is not restricted to periodic sources and linearity allows it to be used to predict the latency of other types of sources as well. A more elaborate discussion is given by Moreira [30].

The latency computation can also be generalized to switching scenarios in SADF, where given an SADF and an automaton defining the possible scenario sequences, we can compute the exact worst-case latency. Details are beyond the scope of this chapter and can be found in [1]. Acceptance conditions on the automaton can be additionally taken into account by identifying and excluding from the analysis any states of the automaton that cannot be reached in any accepting word.

Throughput analysis is also possible on the state space of an SADF. The worst case throughput can be determined from the worst case, the Maximum Cycle Mean, of the delay values on the edges, or the Maximum Cycle Ratio when rewards are considered. In the state space of Figure 7, this is the cycle indicated in bold, corresponding to the alternating execution of scenarios a and b. Note that it has the same cycle mean (3) as the MPAG in Figure 8. Typically, one prefers the analysis on the MPAG though, because the MPAG is usually, as in this example, smaller than the state space. The state space does have the advantage that one can also determine the best case throughput, in the sense of the highest throughput one could guarantee under the assumption that one has full control over the sequence of scenarios that occur. This may be the case, for instance, when the scenarios represent decisions made by a scheduler as in the work of Yang et al. [49], or a supervisory controller as the work of Van der Sanden et al. [36, 35]; see also Section 4.3 in Chapter 9.

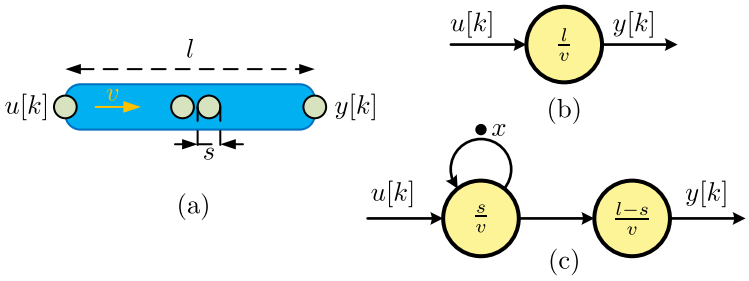

Fig. 10 Example of a conveyor belt and its dataflow linear model

This is achieved by finding a cycle in the state space with a *Minimum* Cycle Mean (or Ratio). One of the possible optimal scenario sequences is indicated in Figure 7 with dashed arrows and circles. Note that such an optimal-throughput scenario sequence cannot be determined from the MPAG. In particular, the Minimum Cycle Mean in the MPAG equals 1, which does not correspond to the optimal throughput cycle in the state space, which has a cycle mean of $5/2$. This can be explained as follows. The MPAG represents all dependencies between individual tokens across different scenarios. The worst-case cycle in this graph simultaneously identifies the worst case scenario sequence and the worst case among all token dependencies. The best-case scenario sequence in the state space still needs to consider the worst-case token dependencies for its throughput. This combination of best-case scenarios and worst-case token dependencies cannot be identified in any cycles in the MPAG.

The optimal throughput solution can also be generalized to the situation that the transitions on the FSA can be partitioned into transitions that are controllable and transitions that are uncontrollable. In that case, optimal attainable throughput can be analyzed by finding optimal strategies in a two-player cycle mean game (or a cycle ratio game in case of rewards) [35].

3.4 Modeling Switched Max-Plus Linear Systems

The Scenario-Aware Dataflow model is an instance of a *switched linear system* in the max-plus linear algebra, i.e., a system that switches between different linear behaviors. Many other models fall in the same category and similar techniques can be applied for their performance analysis. An example is the activity model introduced in Chapter 9. To fit with the switched linear model, the systems needs to adhere to linear modes of operation.

Example 1. Consider a conveyor belt of length l that moves at a constant speed v as shown in Figure 10(a). It can transport objects. For simplicity we ignore the physical dimensions of the objects, so they can be arbitrarily close on the belt and

can be placed on the belt at any time. We model the belt as a system B that takes a (possibly infinite) sequence of input events $u[k]$, which are the points in time at which object k is placed on the start of the belt. The outputs of the system B, $y[k]$, are the time points at which object number k reaches the end of the belt. It satisfies the following equation.

$$y[k] = u[k] + \frac{l}{v}$$

For a system to be linear, it needs to have two properties. It should be *additive* and it should be *homogeneous*. Both are to be interpreted in terms of max-plus algebra.

We assume a system S with a scalar input and a scalar output. The notation and definitions generalize straightforwardly to systems with multiple inputs and outputs and to sequences of inputs or outputs. We use $S(x)$ to denote the time stamp of the output event of system S in response to an input event with time stamp x. Max-plus additivity means that a system satisfies the following rule for all x_1 and x_2:

$$S(\max(x_1, x_2)) = \max(S(x_1), S(x_2)) \ .$$

A more intuitive term for max-plus additivity is *monotonicity* as this is equivalent to the following condition:

$$x_1 \leq x_2 \Rightarrow S(x_1) \leq S(x_2) \ .$$

An important consequence of additivity / monotonicity is that it allows the superposition principle to be applied [13]. This principle states that the response due to the sum of a number of inputs (in our setting the maximum of inputs) can be determined as the sum of the outputs that are due to each of the inputs individually. An example of the principle is given later in this section.

It is easy to see that the belt of Example 1 is additive.

$$B(\max(x_1, x_2)) = \max(x_1, x_2) + \frac{l}{v} = \max(x_1 + \frac{l}{v}, x_2 + \frac{l}{v}) = \max(B(x_1), B(x_2))$$

The second property required for a system to be max-plus linear, is homogeneity. For a max-plus linear system this means that the following must hold for all x and c.

$$S(c + x) = c + S(x)$$

A more intuitive term for max-plus homogeneity is *shift-invariance*, if the time stamp of the input event is shifted by an amount c then the time stamp of the output event is shifted by the same amount.

The belt is also easily seen to be shift-invariant.

$$B(c + x) = (c + x) + \frac{l}{v} = c + (x + \frac{l}{v}) = c + B(x)$$

Hence, the belt is max-plus linear. It has no internal state, so its canonical representation consists of only the **D** matrix, which, because there is only one input and

only one output, is just a scalar number with the value $\mathbf{D} = {}^{l}\!/\!_{v}$. The belt also has a, very simple, dataflow representation as a single actor, as shown in Figure 10(b). Figure 10(c) additionally shows a dataflow model of a belt where we do take the physical size of objects into account. We assume it is s. Now the system has internal state x to remember how much space/time the previous object takes. The canonical representation is as follows. Its derivation is left as an exercise for the reader.

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \begin{bmatrix} {}^{s}\!/\!_{v} & {}^{s}\!/\!_{v} \\ {}^{l}\!/\!_{v} & {}^{l}\!/\!_{v} \end{bmatrix}$$

For this system with internal state it is interesting to consider using the superposition principle to determine the output sequence that is the response to the input sequence with $u[0] = 4$ and $u[1] = 5$. We assume that ${}^{l}\!/\!_{v} = 10$ and ${}^{s}\!/\!_{v} = 2$. The input sequence u can be seen as $u = u_0 \oplus u_1$, where u_0 considers only the first input, i.e., $u_0[0] = 4$ and $u_0[1] = -\infty$, and u_1 considers only the second input, i.e., $u_1[0] = -\infty$ and $u_1[1] = 5$. The superposition principle predicts that if y is the output corresponding to u and y_0 and y_1 are the outputs corresponding to u_0 and u_1, respectively, then $y = y_0 \oplus y_1$. Assuming $x[0] = 0$, we have according to Equation 1:

$$\begin{bmatrix} x[1] \\ y_0[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[0] \\ u_0[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \\ 14 \end{bmatrix}$$

$$\begin{bmatrix} x[2] \\ y_0[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[1] \\ u_0[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 6 \\ -\infty \end{bmatrix} = \begin{bmatrix} 8 \\ 16 \end{bmatrix}$$

Hence, $y_0[0] = 14$ and $y_0[1] = 16$. In the same way we can compute y_1:

$$\begin{bmatrix} x[1] \\ y_1[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[0] \\ u_1[0] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 0 \\ -\infty \end{bmatrix} = \begin{bmatrix} 2 \\ 10 \end{bmatrix}$$

$$\begin{bmatrix} x[2] \\ y_1[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} x[1] \\ u_1[1] \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 10 & 10 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 7 \\ 15 \end{bmatrix}$$

Therefore, $y_1[0] = 10$ and $y_1[1] = 15$. We combine both results to conclude that $y[0] = y_0[0] \oplus y_1[0] = 14$ and $y[1] = y_0[1] \oplus y_1[1] = 16$. This can easily be verified to be the correct result in which the first input takes exactly 10 time units (the length of the belt, ${}^{l}\!/\!_{v}$) from input to output, but the second input on the belt is delayed by the first, because they need to be two time units (${}^{s}\!/\!_{v}$) apart. Despite the fact that the two inputs 'interact', their responses can be computed individually and then combined.

Note that the system implementation does not necessarily need to be linear, it only needs to have a linear abstraction in terms of the abstraction relation discussed in Section 2.2. An example is the response time of a job scheduled on a processor. If the arrival time of the job is the input event and its completion time is the output event, then the scheduler is often not shift-invariant. It may still have a linear abstraction. For instance, when the scheduler has a worst-case response time for the job. In that case, the model that adds the worst-case response time to the job arrival time is a linear abstraction of the job scheduled on the processor.

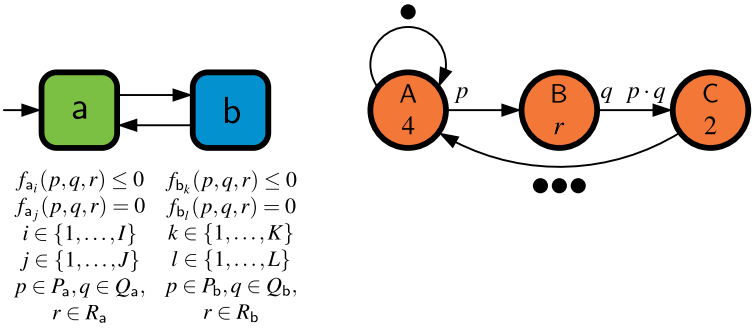

Fig. 11 Example of a parameterized SADF.

3.5 Parametric Analysis

For the worst-case throughput analysis technique explained above to be applicable, the actor firing delays and rates characterizing SDF scenarios must be *fixed* and *known at design-time*. If we deem actor firing delays and rates as *parameters*, then assigning values to those parameters yields an SADF that is amenable to the analysis described.

If we have a system with a number of possible values for each of the parameters, to perform the analysis as described previously in this chapter, we will consequently need to generate a scenario for every possible combination of assignments. Hence, the number of scenarios may get to a point where it will experience compactness [5] or succinctness-related [46] problems. On the analysis side, the product set cardinality hampers the use of SADF in the analysis of systems exposing high levels of data-dependent dynamics in a way that it will render the analysis run-time prohibitive because of the shear number of scenarios that need to be considered. The problem becomes even more intricate if parameters are dependent and not all combinations of values can occur. The dependencies may be specified explicitly in the construction of the model, e.g., one parameter is given as an expression of another or implicitly, e.g., a dataflow scheduler synthesizes some parameter values [33].

We can address these problems by combining the finite control of SADF with parameterized dataflow into a construct we refer to as parameterized SADF. In particular, we model each scenario using a parameterized dataflow graph.

Parameterization, as a syntactic construct, allows us to represent vast sets of scenarios in a compact way (parameters help keep the size of the model manageable) and to explicitly represent the dependencies between parameters as constraints. The underlying parametric analysis enables us to avoid the enumeration of the parameter product set. An example of a parameterized SADF graph is shown in Figure 11. The dataflow graph on the right hand side of the figure reveals three parameters: p, q and r. Parameters p and q attain values from the set of non-negative integers and are used to parameterize rates of actors A, B and C, while parameter r attains val-

ues from the set of non-negative real numbers and is used to parameterize the firing delay of actor B. The FSA in the left part of the figure indicates that the structure involves two scenarios a and b. Parameter dependencies and parameter bounds are specified per scenario in terms of *scenario domains* described (in the most general case) via:

- a system of non-linear inequalities

  $$f_{\mathsf{a}i}(p,q,r) \leq 0$$

  where $i \in \{1,\ldots,I\}$, $p \in P_{\mathsf{a}} \subset \mathbb{N}_0$, $q \in Q_{\mathsf{a}} \subset \mathbb{N}_0$, $r \in R_{\mathsf{a}} \subset \mathbb{R}_{\geq 0}$, and

  $$f_{\mathsf{b}k}(p,q,r) \leq 0$$

  where $k \in \{1,\ldots,K\}$, $p \in P_{\mathsf{b}} \subset \mathbb{N}_0$, $q \in Q_{\mathsf{b}} \subset \mathbb{N}_0$, $r \in R_{\mathsf{b}} \subset \mathbb{R}_{\geq 0}$
- and a system of non-linear equalities

  $$f_{\mathsf{a}\,j}(p,q,r) = 0$$

  where $j \in \{1,\ldots,J\}$, $p \in P_{\mathsf{a}}$, $q \in Q_{\mathsf{a}}$, $r \in R_{\mathsf{a}}$, and

  $$f_{\mathsf{b}l}(p,q,r) = 0$$

  where $l \in \{1,\ldots,L\}$, $p \in P_{\mathsf{b}}$, $q \in Q_{\mathsf{b}}$, $r \in R_{\mathsf{b}}$.

Each transition of the scenario FSA incurs the invocation of an arbitrary instance of the parameterized scenario that the transition destination state corresponds to. An instance of a parameterized scenario is a concrete scenario obtained by assigning all parameters with values inside the parameterized scenario domain, i.e., values that satisfy the constraints. The operational semantics of the model is illustrated in Figure 12. For illustrative purposes, scenario domains are depicted as *2-D* planes (recall that a domain can be non-linear too) in the $p-q-r$ space. For example, whenever a transition $\mathsf{a} \rightarrow \mathsf{b}$ is taken, a concrete scenario obtained by assigning values from the b scenario domain to parameters p, q and r, is executed. Example of such assignments are depicted by the points $(p_{\mathsf{a}}, q_{\mathsf{a}}, r_{\mathsf{a}})$ and $(p_{\mathsf{b}}, q_{\mathsf{b}}, r_{\mathsf{b}})$.

Throughput analysis for parameterized SADF is based on the max-plus switched linear system semantics of SADF. In particular, starting from a set of parameterized scenarios, for each of them, a single representative max-plus matrix is generated that captures the worst-case system behavior per parameterized scenario. The matrix is derived by solving a series of non-linear optimization problems. Thereafter, the matrices are used, in the same way, to construct the corresponding MPAG the MCM of which defines the inverse of the worst-case throughput. Experimental results indicate that the approach is advantageous for both modeling and analysis perspective for graphs with broad-ranging interdependent parameters. More detailed explanations can be found in [40, 41].

The concept of parameterized SADF presented above treats the case where parameters are not fixed during the execution of the systems. In particular, parameters are allowed to change from one invocation of a scenario to the next invocation of the

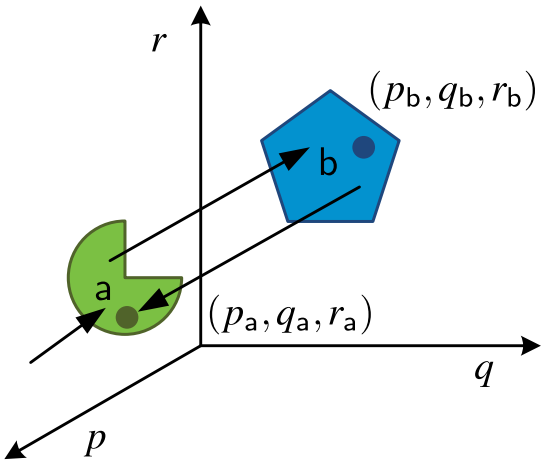

Fig. 12 Operational semantics of the parameterized SADF of Figure 11.

same scenario. The analysis results in a greatest lower bound of the performance of any possible scenario sequence and parameter valuations. However, we may often encounter systems where parameters do not change during the system execution, or change only infrequently. For such a parameterized SADF, instead of a single worst-case performance result, we can find throughput expressions that present the throughput as a function of the scenario parameter values (actor firing delays and actor port rates). Calculation of throughput for a particular parameter valuation is then merely an evaluation of this function for the specific parameter values, which is much faster than the standard throughput analysis. This result may be particularly important for run-time-management, which we discuss at the end of the chapter.

We first consider the case where actor firing delays can be parameters. This setting was first discussed in [18] for SDF and in [9] for SADF. Consider the example SDF shown in Figure 13. The scenario FSA is shown in the left part of the figure,

	parametric execution time		
scenario	A_0	A_1	A_2
a	$6p_1$	$2.5p_2$	$2p_2$
b	$1p_1$	$4.5p_2$	$1p_2$

parameter range
$1 \leq p_1 \leq 5$
$1 \leq p_2 \leq 5$

Fig. 13 An SADF with parametric actor firing delays

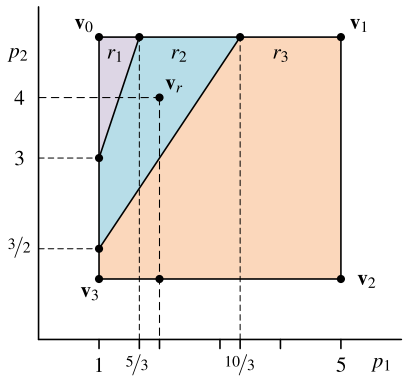

Fig. 14 Throughput regions of the parameterized SADF of Figure 13.

while the scenario graph is in the right part of the figure. By now a reader will be able to deduce that the model involves two scenarios: a and b. In each scenario, actors A_0, A_1 and A_2 perform one firing each. The firing delays are functions of two parameters, p_1 and p_2, as specified in the table in the lower part of Figure 13.

When the actor firing delays are given as linear expressions of parameters, it can be shown that parts of the parameter space that share the same MCM expression (recall that the inverse of the MCM defines the throughput) form convex polyhedra. We call these convex polyhedra *throughput regions*.

For the example SDF of Figure 13, the throughput regions are shown in Figure 14. Values of the parameter p_1 span the x-axis, while the values of p_2 span the y-axis. The model has three throughput regions denoted r_1, r_2 and r_3 with the following expressions for the MCM μ:

- $\mu = 5.5p_2$ if $3p_1 \leq p_2$ for $(p_1, p_2) \in r_1$;
- $\mu = 3p_1 + 4.5p_2$ if $p_2 \leq 3p_1 \leq 2p_2$ for $(p_1, p_2) \in r_2$ and
- $\mu = 6p_1 + 2.5p_2$ if $3p_1 \geq 2p_2$ for $(p_1, p_2) \in r_3$.

The inverses of MCM expressions define the throughput expressions. For more details we refer the reader to [9].

The approach we just presented is only applicable to graphs with parameterized actor firing delays. Furthermore, these delays are constrained to be linear combinations of parameters. Indeed, the key assumption in [9] is linearity. Introduction of parameterized rates implies non-linearity as products of rates may appear in MCM expressions. Therefore, the results of [9] were generalized in [38], which can deal with graphs of certain structure involving both parameterized rates and actor firing delays. The technique linearizes the problem by expanding it into a high-dimensional space. Unfortunately, although theoretically relevant, the technique is of limited usability (limited to a set of only a few critical parameters) because manipulation of high-dimensional polytopes incurs a high penalty in performance.

Listing 1 Pseudo-code of an abstract video decoder.

```
1: frame = buffer_frame()
2: if detect_frame_type(frame) == full then
3:   x = decode_full(frame)
4:   sub = subtitle_overlay(x)
5: else
6:   x = decode_delta(frame)
7: end if
8: output = construct_frame(x)
9: display(output, sub)
```

4 A programming model for SADF

SADF is a dataflow model that can represent scenarios of pipelined applications. It can also be used as a programming paradigm for dynamic pipelined applications in which the dynamic behavior can be expressed as different modes of operation. An additional advantage of such an approach is that the application structure can directly guide the process of design-time scenario identification. This section describes such a programming model based on SADF and its realization on the CompSOC platform [25, 26]. It explicitly addresses the challenge of run-time scenario identification and the required switching and reconfiguration to execute the corresponding behavior.

4.1 A Scenario-aware Dataflow Application

We consider streaming real-time applications that process data when it is received. The control flow of such applications often depends on the received data. Upon receiving a video frame, for example, a decoder detects if it is a full frame or a delta frame and invokes the appropriate decoding function. In a sequential language, a programmer may solve such a dependency with a simple *if-else* construct as shown in Listing 1. This likely leads to the identification of both cases as separate scenarios.

SADF is a natural way to describe the behavior of this application and captures different input-dependent control flows in its *scenarios*. Every possible control flow is captured in a *scenario graph* by the programmer, possibly based on existing sequential code. The scenario graph for decoding a full video frame (S_{full}) is depicted in Figure 15. When a delta frame is detected, the application behavior and thus also the scenario graph are different, namely S_{delta} in Figure 16. Actor names are abbreviations of the functions in Listing 1. The possible scenario sequences are specified by the FSM in Figure 17.

The SADF model allows tight analysis of applications with input-data dependent control flow, it does not define an *implementation* model. Consider the video decoder, where bf and dft are always executed first in either scenario. Only after executing `detect_frame_type` (dft) the next scenario has been detected and

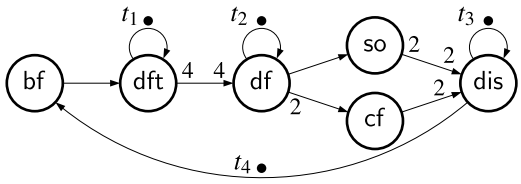

Fig. 15 Scenario graph, S_{full}, for decoding a full video frame.

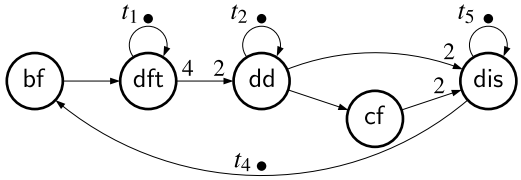

Fig. 16 Scenario graph, S_{delta}, for decoding a delta video frame.

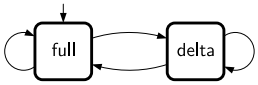

Fig. 17 The FSM of the video decoder with scenarios full and delta.

the control diverges between the scenarios. At run-time it is impossible to decide which of the scenarios is being executed until after it has started. Thus, a causality dilemma is encountered if we want to use the scenario model as an implementation model or a programming paradigm.

Because scenario graphs capture different behaviors of the same application, we argue that a given number of actors and tokens at the start of each scenario are common to all scenarios. In the example, these are actors bf and dft and tokens t_1 and t_4. After executing this maximal prefix graph the current scenario is assumed to be known. This prefix graph must be marked as such by the programmer after which it is automatically split off in a *detector scenario* det, see Figure 18 (the additional actors, $\mathsf{Sel}_{\mathsf{det}}$ and $\mathsf{Sw}_{\mathsf{det}}$, and additional tokens will be explained later). After execution of S_{det} the next scenario is known and can be executed, e.g. S_{full} depicted in Figure 19. This solves the causality dilemma and is explained in detail in Section 4.2.

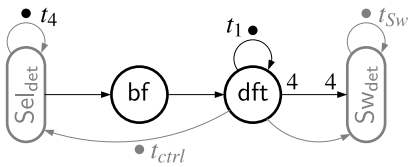

Fig. 18 Analysis graph, S_{det}, of the detector scenario det.

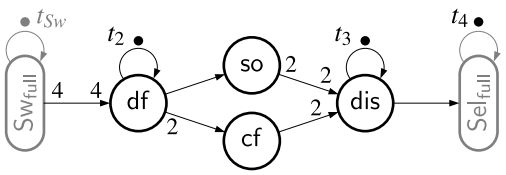

Fig. 19 Analysis graph, S_{full}, of the full frame scenario.

4.2 Sequence Analysis

In Section 4.1 we introduced a solution to the scenario detection causality dilemma by splitting off the detector scenario det from the original scenarios, see Figure 18. We assume the identification of the detector subgraph is done by the programmer. All the following steps are automated. First the FSM is transformed to execute scenario det before each transformed original scenario, see Figure 20.

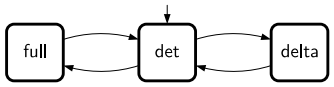

Fig. 20 The extended FSM with detector scenario det.

To model the transport of tokens from and to the detector scenario, we use *switch* and *select* actors borrowed from boolean dataflow [5]. These are instantiated on the outgoing channels (switch) and incoming channels (select) at which the scenario graphs are split, and assigned a worst-case execution time (WCET) of zero. See Figures 19 and 21. The switch and select actors receive boolean control tokens from the dft frame detector. Each Sw and Sel receives a self-edge with a synchronization token that has the same label in every scenario. Token t_4 is a special case. It is available for the next scenario only after the dis actor has finished its firing and it is consumed by the bf actor of the following det scenario. Therefore t_4 is moved onto the self-edge of Sel actors of the scenario graphs, removing the need for an additional synchronization token. Additionally, we need one initial control token t_{ctrl} on the edge to the initial select actor of the detection scenario to make the graph deadlock free. The analysis graphs thus generated are shown in Figures 18, 19 and 21. The Sw and Sel actors as well as newly inserted channels and tokens actors are indicated in grey.

Temporal analysis of this video decoder with SDF^3 [43] visits the scenarios as indicated by the FSM, starting with scenario det. It can be shown that the throughput of the modified graph is identical to the throughput of the original graph [26]. In each new scenario graph it is known from the start of the execution what the current scenario is, which allows the scenario graph to be directly implemented.

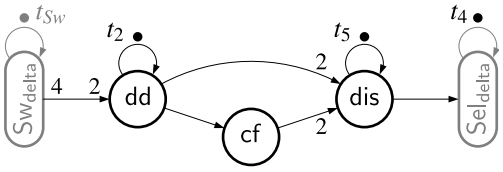

Fig. 21 Analysis graph, S_{delta}, of the delta frame scenario.

4.3 Scenario Execution

In this section we present an implementation for executing a sequence of scenarios that solves the following practical aspects: (*i*) *switch* and *select* implementation; (*ii*) extending static-order actor schedules on-the-fly.

4.3.1 Switch and Select Implementation

During analysis the *switch* and *select* actors ensure synchronization but function as regular SDF actors. While scenarios are analysed separately, our implementation glues $\mathsf{S}_{\mathsf{det}}$ to the graphs of the other scenarios. In practice the switch acts as multiplexer and the select as de-multiplexer. For execution there is *one* detector subgraph serving both $\mathsf{S}_{\mathsf{full}}$ and $\mathsf{S}_{\mathsf{delta}}$ with tokens, see Figure 22. Actors Sw and Sel are indicated in grey. Synchronization token t_{Sw} is not relevant for execution.

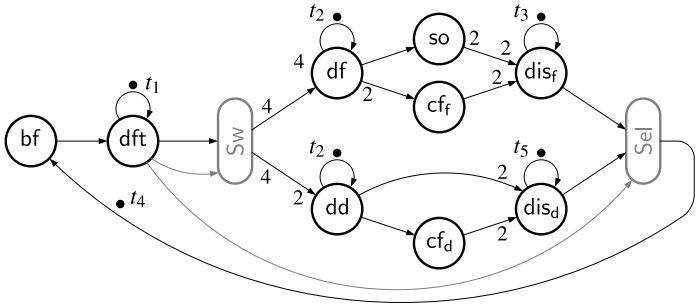

Fig. 22 The merged execution graph, the *switch* and *select* actors are indicated in grey.

We propose a solution that exploits the `libFIFO` [19] library, that implements FIFO buffers that can be disconnected and reconnected without invalidating data. A *switch* actor will have only a single output port, to which the proper channel is connected depending on the detected scenario (see Figure 23). This swapping of FIFO channels is indicated with the symbol for an electrical switch, which connects the single output port either to the channel to actor df or the channel to dd. The other channel is left unconnected on one end, effectively giving it rate zero. The FIFO

swap must take place before Sw fires. *Select* actors are similar but demultiplex two channels to one.

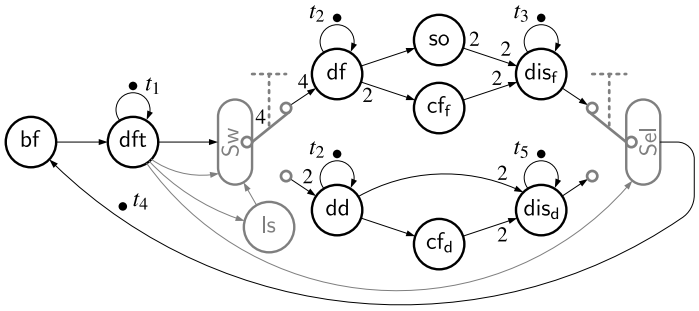

Fig. 23 The merged execution graph with implementation details, the *switch/select* and *load schedule* (ls) actors for a one-processor mapping are indicated in grey. The swapping of FIFO channels is indicated with the electrical symbol for a switch, also in grey. Note that this is not a valid dataflow graph.

4.3.2 Extending Static-Order Schedules

We schedule actors on a processor with the `libDataflow` library [19] that executes dataflow graphs by iterating over a given static-order (SO) schedule. Such a schedule can be produced with SDF^3 [43]. The schedule blocks if an actor is not ready to fire. However, the SO schedule of our proposed solution changes depending on the detected scenario. Therefore we use a new scheduling concept that we dub the *rolling static-order* (RSO) scheduler [26].

Execution starts with $\mathsf{S}_{\mathsf{det}}$, so if we were to map the decoder to a single processor the SO schedule starts with $[\mathsf{bf}, \mathsf{dft}, \mathsf{Sw}]$. After firing dft the next scenario is known and the SO schedule can be extended. If scenario full is detected, then the sequence $[\mathsf{df}, \mathsf{cf_f}, \mathsf{cf_f}, \mathsf{so}, \mathsf{dis_f}, \mathsf{Sel}, \mathsf{bf}, \mathsf{dft}, \mathsf{Sw}]$ is concatenated to the "rolling" schedule. Note that Sel is the last actor in the schedule of $\mathsf{S}_{\mathsf{full}}$, and we immediately concatenate the next detector scenario. This ensures that the scheduler will never run out of actors to schedule. A multi-processor mapping works similarly, the only constraint we impose is that a *switch* must be mapped onto the same processor as the actor preceding it.

The RSO scheduler has been implemented in `libDataflow` for the CompSOC platform. We initialize it with a unique SO schedule for each scenario; the start is set to $\mathsf{S}_{\mathsf{det}}$. An additional *load schedule* (ls) actor is inserted right after dft *on every processor*. See Figure 23 for the single-processor example. These ls actors receive a scenario token from dft and extend the schedule accordingly. The example schedule of $\mathsf{S}_{\mathsf{det}}$ changes to $[\mathsf{bf}, \mathsf{dft}, \mathsf{ls}, \mathsf{Sw}]$. We furthermore exploit the ls actor to connect all FIFOs correctly before the actors Sw or Sel fire. This dependency is visualized with

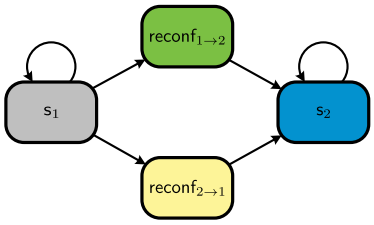

Fig. 24 Automaton defining the reconfiguration scenario sequences

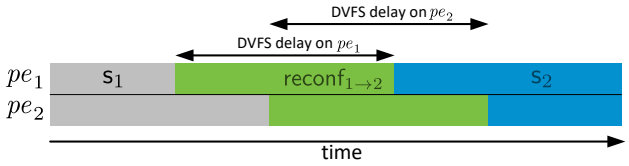

Fig. 25 Gantt chart of the reconfiguration

a grey channel in Figure 23. Details on the implementation and an experimental evaluation can be found in [26].

5 Run-Time Methods

This section briefly discusses some methods that can be applied for resource management using the SADF model at run-time.

5.1 Run-time Management

An execution platform may support for execution parameters to be changed or selected at run-time, depending on run-time state, like resource availability or variations in deadlines or slack. Such settings can be considered as run-time situations that can be characterized and represented by scenarios and correspondingly modelled with dataflow models.

The selection of different settings may lead to run-time opportunities to optimize multiple aspects of the running system. Run-time situations can be seen as partially determined by the environment, e.g., by load induced by different kinds of application input, and partially by the system's run-time management. [49] shows how the interplay between environment and run-time management can be seen as a two-player game, where one player is the run-time management and the opponent player is the environment. The moves in the game can be modelled as dataflow

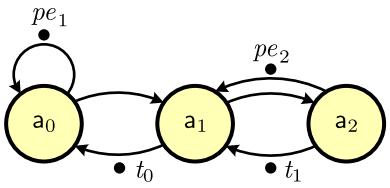

Fig. 26 Dataflow graph of the application scenario

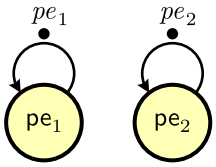

Fig. 27 Dataflow graph of the DVFS reconfiguration scenario

scenarios. In this case known results from the theory of two player games, in particular, mean-payoff games and ratio-games, can be exploited to determine optimal positional strategies. This leads to an optimal run-time strategy under environmental variations.

We consider a simple example taken from [8]. It considers an application shown in Figure 26. The dataflow graph of the application consists of three actors, a_0, a_1 and a_2 with data dependencies captured by the tokens t_0 and t_1. It is additionally assumed that there are two processing elements, pe_1 and pe_2 and that pe_1 executes actor a_0 and pe_2 executes actors a_1 and a_2. The mapping of tasks to processing elements is modeled with the tokens pe_1 and pe_2 that capture the availability time of the processing resources pe_1 and pe_2 respectively. The processing elements support DVFS [6] to exploit a trade-off between processing speed and power consumption. We assume that there are two settings to consider, having a specific DVFS setting for each of the processing elements. We represent these two settings by two scenarios, s_1 and s_2. Both scenarios follow the application graph of Figure 26, but with different actor execution times. Run-time switching between s_1 and s_2 is possible, but changing the DVFS setting for a processing element takes time and energy. The switching process is therefore modeled by a dataflow scenario itself. The graph is shown in Figure 27. Note that the graph expresses that a DVFS change can only start after the processing element is available (has finished the workload of previous scenarios) and then takes some amount of time (the execution time of the actor), after which the processing element is available for processing of new workload, expressed by the production time of the token. Note that the scenario defines that both processing elements need to reconfigure, but there are no dependencies between them. They do not need to happen at the same moment, but can follow the pipelined execution of the application. We assume the reconfigurations may take different amounts of time and therefore there are two switching scenarios $\mathsf{reconf}_{1\to 2}$

and $\mathsf{reconf}_{2\to1}$. The possible sequences of scenarios are defined by the automaton in Figure 24. It specifies that a change of DVFS mode needs to invoke the corresponding reconfiguration scenario.

Figure 25 shows a Gantt chart of a fragment of the behavior including a DVFS switch from scenario s_1 to scenario s_2 by the intermediate reconfiguration scenario $\mathsf{reconf}_{1\to2}$. The reconfiguration follows the pipelined execution of the application and pe_1 reconfigures before pe_2.

The SADF model defines all the possible behaviors that a run-time manager can choose to execute with this application. Those behaviors correspond to scenario sequences of the SADF model. The model predicts the worst-case performance of such a behavior. This can be used by an optimization strategy to find the optimal behavior in terms of power consumption that satisfies the performance constraints, for example using Model Predictive Control techniques.

Parametric analysis, as discussed in Section 3 can be particularly interesting for run-time situations. Actor execution times may only be available at run-time, but there may not be enough time or resources for a full dataflow analysis algorithm to be executed at run-time. In such a case the parametric analysis results can be applied to quickly obtain the analysis result in a particular run-time situation when the parameter values have become known. Furthermore, the parametric analysis can be utilized to achieve energy savings in a setting with varying throughput requirement. Imagine a situation where one can quantify the Quality of Service (QoS) via throughput. In most cases, the higher the throughput, the higher QoS and vice-versa. On the other hand, achieving a higher or lower throughput, in terms of dataflow, means that the actors will have to attain shorter or longer firing delays, respectively. The question is, given a run-time change in QoS expressed as a throughput constraint, how long is short enough to meet the throughput constraint. [8] discusses a heuristic strategy to find such optimal Dynamic Voltage and Frequency Scaling (DVFS) settings using the parametric throughput expression of the model of the application. In Chapter 9 is it shown how this strategy can also be applied to optimal controller synthesis problems for flexible manufacturing systems.

6 Conclusions

This chapter has discussed how the scenario methodology can be applied to and combined with dataflow modelling and analysis. The resulting model SADF is an expressive model that can express variation in functional behavior and consequently diverse run-time situations. Dataflow models can also be used in the context of real-time streaming applications as a formal abstraction of concrete realizations that guarantees preservation of performance for its refinements.

We have detailed the model and its semantics and have described common performance analysis methods for throughput and latency, including parametric analysis techniques. Timed dataflow is a performance oriented model in a wider class of max-plus linear systems. The properties of max-plus linear systems have been

discussed to better understand the essential properties of systems to be amenable to being modeled with max-plus linear abstractions.

We have also shown how the SADF model can be applied as a programming model to ensure a good match between an implementation on a multiprocessor platform and the abstract dataflow model. It wa shown how to integrate the scenario detection methods and derive an accurate model of a causal implementation of the behavior. An online static-order scheduling technique is presented to realize the implementation.

We have additionally illustrated how the dataflow model and the parametric analysis can be combined to realize efficient run-time management strategies.

Acknowledgements This research is supported in part by the ARTEMIS joint undertaking through the ALMARVI project (621439) and by the ITEA3 project 14014 ASSUME.

References

1. Alizadeh Ara, H., Behrouzian, A., Hendriks, M., Geilen, M., Goswami, D., Basten, T.: scalable analysis of multi-scale dataflow models. ACM Trans. Embedded Computing Systems **16**(4), 80:1–80:26 (2018)
2. Baccelli, F., Cohen, G., Olsder, G., J.P.Quadrat: Synchronization and Linearity. John Wiley & Sons (1992)
3. Battacharyya, S.S., Lee, E.A., Murthy, P.K.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers, Norwell, MA, USA (1996)
4. Bijlsma, T., Bekooij, M., Smit, G.: Circular buffers with multiple overlapping windows for cyclic task graphs. In: P. Stenström (ed.) Transactions on High-Performance Embedded Architectures and Compilers III, Lecture Notes in Computer Science. Springer Verlag (2011). DOI 10.1007/978-3-642-19447-4
5. Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, EECS Department, University of California, Berkeley (1993)
6. Chandrakasan, A.P., Sheng, S., Brodersen, R.W.: Low-power cmos digital design. IEEE Journal of Solid-State Circuits **27**(4), 473–484 (1992). DOI 10.1109/4.126534
7. Cochet-Terrasson, J., Cohen, G., Gaubert, S., Gettrick, M., Quadrat, J.P.: Numerical computation of spectral elements in max-plus algebra. In: Proc. of the IFAC Conference on System Structure and Control. Nantes (1998)
8. Damavandpeyma, M., Stuijk, S., Basten, T., Geilen, M., Corporaal, H.: Throughput-constrained dvfs for scenario-aware dataflow graphs. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 175–184 (2013). DOI 10.1109/RTAS.2013.6531090
9. Damavandpeyma, M., Stuijk, S., Geilen, M., Basten, T., Corporaal, H.: Parametric throughput analysis of scenario-aware dataflow graphs. In: 2012 IEEE 30th International Conference on Computer Design (ICCD), pp. 219–226 (2012). DOI 10.1109/ICCD.2012.6378644
10. Dasdan, A., Gupta, R.K.: Faster maximum and minimum mean cycle algorithms for system-performance analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **17**(10), 889–899 (1998). DOI 10.1109/43.728912
11. Dhingra, V., Gaubert, S.: How to solve large scale deterministic games with mean payoff by policy iteration. In: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools, valuetools '06. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1190095.1190110. URL http://doi.acm.org/10.1145/1190095.1190110

12. Gaubert, S.: Performance evaluation of (max, +) automata. IEEE Trans. Automatic Control **40**(12), 2014–2025 (1995)
13. Geilen, M.: If we could go back in time... on the use of 'unnatural' time and ordering in dataflow models. In: M. Lohstroh, P. Derler, M. Sirjani (eds.) Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday, pp. 267–286. Springer International Publishing, Cham (2018). DOI 10.1007/978-3-319-95246-8_16. URL https://doi.org/10.1007/978-3-319-95246-8_16
14. Geilen, M., Basten, T.: Requirements on the execution of Kahn process networks. In: P. Degano (ed.) Proc. Of the 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003. LNCS Vol.2618. Springer Verlag, Berlin (2003)
15. Geilen, M., Falk, J., Haubelt, C., Basten, T., Theelen, B., Stuijk, S.: Performance analysis of weakly-consistent scenario-aware dataflow graphs. Journal of Signal Processing Systems **87**(1), 157–175 (2017). DOI 10.1007/s11265-016-1193-7. URL http://dx.doi.org/10.1007/s11265-016-1193-7
16. Geilen, M., Stuijk, S.: Worst-case performance analysis of synchronous dataflow scenarios. In: International Conference on Hardware-Software Codesign and System Synthesis, CODES+ISSS 10, Proc., Scottsdale, Az, USA, 24-29 October, 2010, pp. 125–134 (2010)
17. Geilen, M., Tripakis, S., Wiggers, M.: The earlier the better: A theory of timed actor interfaces. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, HSCC '11, pp. 23–32. ACM, New York, NY, USA (2011)
18. Ghamarian, A.H., Geilen, M.C.W., Basten, T., Stuijk, S.: Parametric throughput analysis of synchronous data flow graphs. In: 2008 Design, Automation and Test in Europe, pp. 116–121 (2008). DOI 10.1109/DATE.2008.4484672
19. Goossens, K., Azevedo, A., Chandrasekar, K., Gomony, M.D., Goossens, S., Koedam, M., Li, Y., Mirzoyan, D., Molnos, A., Nejad, A.B., Nelson, A., Sinha, S.: Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow. SIGBED Rev. **10**(3), 23–34 (2013). DOI 10.1145/2544350.2544353. URL http://doi.acm.org/10.1145/2544350.2544353
20. Gu, R., Janneck, J.W., Raulet, M., Bhattacharyya, S.S.: Exploiting statically schedulable regions in dataflow programs. J. Signal Process. Syst. **63**(1), 129–142 (2011). DOI 10.1007/s11265-009-0445-1. URL http://dx.doi.org/10.1007/s11265-009-0445-1
21. Heidergott, B., Olsder, G.J., van der Woude, J.: Max Plus at Work. Princeton University Press (2006)
22. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). DOI 10.1145/359576.359585. URL http://doi.acm.org/10.1145/359576.359585
23. Jantsch, A.: Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
24. Kahn, G.: The semantics of a simple language for parallel programming. In: J. Rosenfeld (ed.) Information Processing 74: Proceedings of the IFIP Congress 74, Stockholm, Sweden, August 1974, pp. 471–475. North-Holland, Amsterdam, Netherlands (1974)
25. van Kampenhout, R., Stuijk, S., Goossens, K.: A scenario-aware dataflow programming model. In: Digital System Design (DSD), 2015 Euromicro Conference on, pp. 25–32 (2015). DOI 10.1109/DSD.2015.28
26. van Kampenhout, R., Stuijk, S., Goossens, K.: Programming and analysing scenario-aware dataflow on a multi-processor platform. In: Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE) (2017)
27. Lee, E., Messerschmitt, D.: Static scheduling of synchronous data flow programs for digital signal processing. Computers, IEEE Transactions on **C-36**(1), 24–35 (1987). DOI 10.1109/TC.1987.5009446
28. Lundqvist, T., Stenström, P.: Timing anomalies in dynamically scheduled microprocessors. In: Proceedings of the 20th IEEE Real-Time Systems Symposium, RTSS '99, pp. 12–. IEEE Computer Society, Washington, DC, USA (1999). URL http://dl.acm.org/citation.cfm?id=827271.829103

29. Moonen, A., Bekooij, M., van den Berg, R., van Meerbergen, J.L.: Practical and accurate throughput analysis with the cyclo static dataflow model. In: 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2007), October 24-26, 2007, Istanbul, Turkey, pp. 238–245 (2007). DOI 10.1109/MASCOTS.2007.52. URL http://dx.doi.org/10.1109/MASCOTS.2007.52
30. Moreira, O.: Temporal analysis and scheduling of hard real-time radios running on a multiprocessor. Ph.D. thesis, Eindhoven University of Technology (2012)
31. Moreira, O., Basten, T., Geilen, M., Stuijk, S.: Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. IEEE Transactions on Computers **59**(2), 188–201 (2010). DOI 10.1109/TC.2009.155. Cited By 24
32. Moreira, O., Corporaal, H.: Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor. Springer (2014)
33. Neuendorffer, S., Lee, E.: Hierarchical reconfiguration of dataflow models. In: Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on, pp. 179–188 (2004). DOI 10.1109/MEMCOD.2004.1459852
34. Parks, T.: Bounded Scheduling of Process Networks. Ph.D. thesis, University of California, EECS Dept., Berkeley, CA (1995)
35. van der Sanden, B.: performance analysis and optimization of supervisory controllers. Ph.D. thesis, Eindhoven University of Technology (2018)
36. van der Sanden, B., Bastos, J., Voeten, J., Geilen, M., Reniers, M.A., Basten, T., Jacobs, J., Schiffelers, R.R.H.: Compositional specification of functionality and timing of manufacturing systems. In: 2016 Forum on Specification and Design Languages, FDL 2016, Bremen, Germany, September 14-16, 2016, pp. 1–8 (2016). DOI 10.1109/FDL.2016.7880372. URL https://doi.org/10.1109/FDL.2016.7880372
37. Siyoum, F., Geilen, M., Eker, J., von Platen, C., Corporaal, H.: Automated extraction of scenario sequences from disciplined dataflow networks. In: Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on, pp. 47–56 (2013)
38. Skelin, M.: Worst-case performance analysis of scenario-aware real-time streaming applications. Ph.D. thesis, Norwegian University of Science and Technology (NTNU) (2016)
39. Skelin, M., Geilen, M.: Compositionality in scenario-aware dataflow: A rendezvous perspective. In: Proc. of LCTES'18 (2018)
40. Skelin, M., Geilen, M., Catthoor, F., Hendseth, S.: Parameterized dataflow scenarios. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **36**(4), 669–682 (2017). DOI 10.1109/TCAD.2016.2597223
41. Skelin, M., Geilen, M., Catthoor, F., Hendseth, S.: Worst-case performance analysis of sdf-based parameterized dataflow. Microprocessors and Microsystems **52**, 439 – 460 (2017). DOI https://doi.org/10.1016/j.micpro.2016.12.004. URL http://www.sciencedirect.com/science/article/pii/S0141933116304094
42. Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors: Scheduling and Synchronization, 2nd edn. CRC Press, Inc., Boca Raton, FL, USA (2009)
43. Stuijk, S., Geilen, M., Basten, T.: SDF^3: SDF For Free. In: Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings, pp. 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA (2006). DOI 10.1109/ACSD.2006.23. URL http://www.es.ele.tue.nl/sdf3
44. Stuijk, S., Geilen, M., Basten, T.: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. IEEE Trans. Comput. **57**(10), 1331–1345 (2008). DOI http://dx.doi.org/10.1109/TC.2008.58
45. Stuijk, S., Geilen, M., Basten, T.: A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In: 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, pp. 548–555 (2010). DOI 10.1109/DSD.2010.31
46. Stuijk, S., Geilen, M., Theelen, B., Basten, T.: Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In: Embedded Computer Systems (SAMOS), 2011 International Conference on, pp. 404–411 (2011). DOI 10.1109/SAMOS.2011.6045491

47. Theelen, B., Geilen, M., Basten, T., Voeten, J., Gheorghita, S., Stuijk, S.: A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: MEMOCODE, pp. 185–194 (2006). DOI 10.1109/MEMCOD.2006.1695924
48. van den Boom, T., De Schutter, B.: Modelling and control of discrete event systems using switching max-plus-linear systems. Control Engineering Practice **14**(10), 1199–1211 (2006). DOI 10.1016/j.conengprac.2006.02.006
49. Yang, Y., Geilen, M., Basten, T., Stuijk, S., Corporaal, H.: Playing games with scenario- and resource-aware sdf graphs through policy iteration. In: 2012 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 194–199 (2012). DOI 10.1109/DATE.2012.6176462

Chapter 9: Scenarios in the Design of Flexible Manufacturing Systems

Twan Basten, João Bastos, Róbinson Medina, Bram van der Sanden, Marc Geilen, Dip Goswami, Michel Reniers, Sander Stuijk, Jeroen Voeten

Abstract Modern high-tech Flexible Manufacturing Systems (FMS) such as lithography systems, professional printers, xray machines and electron microscopes are characterized by an increasingly tight coupling between machine control software and the controlled physical processes. Control software and the design and configuration of FMS have an important impact on system productivity and product quality. Model-based, scenario-based design provides means for guaranteeing and optimizing system productivity while ensuring its proper functioning. We show that abstract system-level activity models, semantically grounded in (max,+) algebra with activities capturing execution scenarios of the FMS, can be used for fast and accurate productivity analysis of FMS in early design phases. The same models can be used for supervisory controller synthesis and optimization, providing safety and performance guarantees in the supervisory control software. Finally, scenario-based, adaptive, pipelined control enables optimization of data-intensive control loops in FMS, which in turn impacts system-level productivity.

T. Basten
Eindhoven University of Technology & ESI, TNO, e-mail: a.a.basten@tue.nl
J. Bastos
Eindhoven University of Technology, e-mail: j.p.nogueira.bastos@tue.nl
R. Medina
Eindhoven University of Technology, e-mail: r.a.medina.sanchez@tue.nl
B. van der Sanden
Eindhoven University of Technology, e-mail: b.v.d.sanden@tue.nl
M.C.W. Geilen
Eindhoven University of Technology, e-mail: m.c.w.geilen@tue.nl
D. Goswami
Eindhoven University of Technology, e-mail: d.goswami@tue.nl
M.A. Reniers
Eindhoven University of Technology, e-mail: m.a.reniers@tue.nl
S. Stuijk
Eindhoven University of Technology, e-mail: s.stuijk@tue.nl
J.P.M. Voeten
ESI, TNO & Eindhoven University of Technology, e-mail: j.p.m.voeten@tue.nl