Scenario-Aware Dataflow: Modeling, Analysis and Implementation of Dynamic Applications

Sander Stuijk[1], Marc Geilen[1], Bart Theelen[2], Twan Basten[1,2]
[1] Eindhoven University of Technology, Department of Electrical Engineering, Eindhoven, The Netherlands
[2] Embedded Systems Institute, Eindhoven, The Netherlands
{s.stuijk, m.c.w.geilen, a.a.basten}@tue.nl, bart.theelen@esi.nl

***Abstract*—Embedded multimedia and wireless applications require a model-based design approach in order to satisfy stringent quality and cost constraints. The Model-of-Computation (MoC) should appropriately capture system dynamics, support analysis and synthesis, and allow low-overhead model-driven implementations. This combination poses a significant challenge. The Scenario-Aware DataFlow (SADF) MoC has been introduced to address this challenge. This paper surveys SADF, and compares dataflow MoCs in terms of their ability to capture system dynamics, their support for analysis and synthesis, and their implementation efficiency.**

I. Introduction

Modern embedded systems, such as smart phones, are often executing multiple streaming applications concurrently. A user may, for example, use a mobile phone to watch a video that is being decoded using an MPEG-4 decoder while an MP3 decoder is used to decode the accompanying audio. The applications may be using an internet connection that requires a software defined radio protocol to download the required bit streams. The user will expect that all these applications have a robust behavior and that their performance is guaranteed [1]. At the same time, the resource usage of these applications should be kept as small as possible in order to save energy and so prolong the lifetime of the battery.

In the architecture domain there is a clear trend to use heterogeneous multi-processor systems-on-chip (MPSoCs) to meet the computation requirements of novel applications at affordable energy cost [2]. Programming these systems is a very challenging task, especially since the interaction between all hardware components needs to be considered in order to provide timing guarantees to the applications [3]. Model-based design approaches (e.g., [4], [5], [6], [7], [8], [9]) are being developed to address this challenge. They model applications using a dataflow Model-of-Computation (MoC). Many of these approaches (e.g., [4], [6], [7], [9]) are based on (homogeneous) synchronous dataflow ((H)SDF) graphs [10], [11], because this model is relatively simple and static. This made it possible to develop many design-time analysis techniques (e.g., [12], [13], [14], [15], [16]), as well as efficient implementations. It is, for instance, possible to derive exact bounds on the storage requirements of an application and to statically schedule the tasks inside an application. As a result, there is almost no run-time overhead when running an application on an MPSoC. However, the (H)SDF MoC abstracts from any dynamic behavior of an application. This may lead to a large overestimation of its resource requirements. The dynamism inside modern multimedia and wireless applications is increasing. Exploiting this dynamic behavior to save resources becomes therefore very important. Several model-based approaches (e.g., [5], [8]) use Kahn Process Networks (KPNs) [17] to model the dynamic behavior of applications. However, relevant properties, such as the minimal storage space needed to avoid deadlock, cannot be determined at design-time [18], [19]. Furthermore, tasks in a KPN cannot be scheduled statically. Hence, a run-time mechanism is needed to detect deadlocks and to schedule tasks and to reallocate the storage space assigned to the application. This causes a considerable implementation overhead compared to (H)SDF-based approaches. Moreover, the lack of design-time analysis techniques makes it difficult to use this MoC to design systems that provide timing guarantees to applications.

The selection of a MoC and a corresponding design approach should consider the expressiveness, analyzability (including synthesizability), and implementation efficiency of the MoC. Some MoCs (e.g., HSDF and SDF) are analyzable and allow an efficient implementation. However, they are not expressive enough to capture the dynamic behavior of modern applications. Other MoCs (e.g., KPN) can capture this dynamic behavior, but only limited analysis techniques exist and they have a large implementation overhead. The Scenario-Aware Dataflow (SADF) MoC [20] has been introduced to take a place between these categories of models. It allows modeling of dynamic behavior, analysis techniques are available, and an efficient implementation can be created. The SADF MoC exploits the scenario-based design approach of Gheorghita et al. [21]. In this approach, the dynamic behavior of an application is viewed as a collection of different behaviors (*scenarios*) occurring in some possible orders. Each scenario by itself is fairly static and predictable in performance and resource usage. SADF exploits the static behavior of scenarios. It models the behavior of each scenario with an SDF graph. The dataflow graphs of different scenarios may differ in all aspects (e.g., communication rates or execution times). This makes it possible to exploit the dynamic behavior of applications to save resources while providing timing guarantees. The static behavior within a scenario also makes it possible to derive an implementation with limited run-time overhead.

This paper provides an overview of the Scenario-Aware Dataflow MoC and the available analysis and implementation techniques. Sec. II provides a short introduction into the MoC. It also explains the choices a designer needs to make to model applications in SADF. Analysis techniques are reviewed in Sec. III. The implementation of applications modeled with an SADF graph are discussed in Sec. IV. Sec. V compares

SADF to other dataflow MoCs based on their expressiveness, analyzability, and implementation efficiency. This comparison shows that SADF offers a good trade-off between all these concerns. Sec. VI briefly discusses a tool set that implements all analysis and implementation techniques discussed in this paper. Sec. VII concludes.

II. Modeling Applications with Scenario-Aware Dataflow

Before introducing the SADF MoC, we briefly discuss the SDF MoC on which SADF is based. The graph in the left part of Fig. 1 is an SDF graph when x is assigned a constant value (for instance, 99). This graph models an MPEG-4 Simple Profile decoder. The nodes, called *actors*, communicate with *tokens* sent from one actor to another over the edges. Actors model application tasks and the edges model data or control dependencies. An essential property of SDF graphs is that every time an actor *fires* (executes) it consumes the same amount of tokens from its input edges and produces the same amount of tokens on its output edges. These amounts are called the *rates* (indicated next to the edge ends, while rates of 1 are omitted for clarity). An actor can only fire if sufficient tokens are available on the edges from which it consumes. Tokens thus capture dependencies between actor firings. Such dependencies may originate from data dependencies, but also from dependencies on shared resources.

The rates determine how often actors have to fire w.r.t. each other such that productions and consumptions are balanced. These rates are constant, which forces an SDF graph to execute in a fixed repetitive pattern, called an *iteration*. An iteration consists of a set of actor firings that have no net effect on the token distribution. These actor firings typically form a coherent collection of computations. An iteration could, for instance, correspond to the processing of a frame in a video stream. This makes iterations the natural granularity for defining scenarios, from the perspective of the application and from the perspective of the model. Note that subsequent iterations are allowed to overlap in time. Hence, different scenarios may be active simultaneously, typically in a pipelined fashion.

The dynamic behavior of an application can be captured in a set of scenarios. Each scenario can be modeled with an SDF graph. A finite state machine (FSM) is added to represent the possible orders in which the active scenarios occur. These SDF graphs together with the FSM, form a model of the application. This set of SDF graphs and the FSM are called an *FSM-based SADF graph*. Consider, as an example, the MPEG-4 decoder of Theelen et al. [20] shown in Fig. 1. The frame detector (FD) models the part of the application that determines the frame type and the number of macro blocks to decode. The decoder model supports two different types of frames (I and P type). When a frame of type I is found, a total of 99 macro blocks must always be processed. This scenario is called '$I99$'. A frame of type P requires processing between 0 and 99 macro blocks. The workload varies considerably depending on the number of macro blocks that is processed. Therefore, a number of different scenarios 'Px' are defined based on the number of macro blocks that must be processed. The graph contains different scenarios for the situations in which (up to) 0, 30, 40, 50, 60, 70, 80, or 99 macro blocks are processed for a single P frame. Within each scenario, the VLD and IDCT operations are performed for every individual block. The other operations are performed once per frame. Therefore, the communication rates vary with each scenario. x is set equal to the maximum number of macro blocks that may need to be processed in the scenario, which can be shown to be conservative in this case. Note that there is a trade-off between the number of scenarios, the run-time of the analysis techniques, and the implementation efficiency. A designer should consider this trade-off when selecting the scenarios and modeling the application.

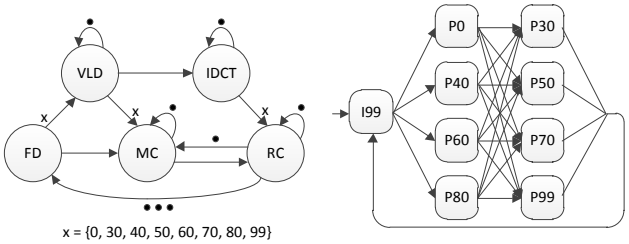

Fig. 1. FSM-based SADF graph of an MPEG-4 SP decoder.

From a syntactical perspective, the FSM-based SADF model is equivalent to the Heterochronous Dataflow (HDF) MoC [22]. This MoC introduces dynamism through a finite state machine that executes one iteration of an SDF graph in each state, causing a transition to a next state. It allows each state to have a different SDF graph. HDF does not have a timed version which can be used for timing analysis. The analysis techniques developed for HDF focus on a sequential execution of HDF graphs. In contrast, techniques developed for the FSM-based SADF MoC focus on timing analysis of parallel executions, as detailed in the next section.

FSM-based SADF/HDF is a restricted form of the general SADF MoC as it was introduced in [20], [23]. Different from FSM-based SADF, general SADF allows scenario changes within an iteration of the graph. This makes it possible to model behavior in which the number of tokens produced and consumed by an actor changes during that scenario. Consider as an example an entropy decoder that reads a variable number of bits from a bitstream before producing a decoded symbol. An FSM-based SADF graph would model this behavior with some worst case abstraction, such as the worst-case number of bits consumed per produced symbol. The general SADF MoC can however distinguish different consumption rates. This may allow it to provide tighter bounds on the resource requirements or timing behavior. The general SADF MoC also extends the FSM with probabilities on scenario occurrences. This turns the FSM into a Markov chain. It actually allows multiple Markov chains to be associated with different actors. This makes it possible to model hierarchical control. It also allows discrete execution time distributions instead of the fixed execution times used in FSM-based SADF. This makes it possible to compute various best/worst-case, (probabilistic) reachability and average-case performance metrics.

III. Analyzing SADF graphs

This section gives an overview of the most important analysis techniques for SADF graphs. Sec. III-A deals with throughput analysis. Sec. III-B reviews latency analysis techniques. Techniques to analyze the buffer requirements of an SADF graph are discussed in Sec. III-C. If implementation aspects such as scheduling decisions need to be taken into account in the analysis, it is assumed that they are modeled into the graph. Methods to do so are beyond the scope of this section. Interested readers are referred to for example [4], [9].

A. Throughput Analysis

Throughput is perhaps the most dominant performance metric used in the realization of applications. Depending on the type of application, one may be interested in the worst-case throughput (i.e., a guaranteed lower bound on the application throughput) or on the expected or long-run average throughput. We discuss the available techniques for both cases separately.

Worst-case throughput analysis. To determine the worst-case throughput of an application modeled with a dataflow graph, we first of all assume that we can find (upper-bounds on) the worst-case execution times of actors. Our assumption on the dataflow models is that these execution times are accurate for the real systems we consider, when the actors execute in a self-timed or data-driven manner. Actors start their firings as soon as they are enabled, i.e., as soon as they have sufficient input tokens available. Effectively they start when the last token they need arrives, their firings take a fixed duration and after that they produce their output tokens. $(\max, +)$-algebra [24] is a very suitable mathematical framework for studying this type of behavior. It is a linear algebra based on the operators $\max$, maximum of two real numbers and $+$, addition, over the set of real numbers extended with $-\infty$. If we think of tokens as labeled with the time at which they become available, then we can see that the starting time of an actor firing is the maximum of the time labels of the tokens it needs. The fixed duration of the firing is obviously expressed by addition and the result of the firing are the new labels of the freshly produced tokens. Moreover, it is known [24] that SDF graphs can be equivalently represented as event graphs, which in turn can be represented by a matrix over the $(\max, +)$-algebra, in such a way that if we think of the initial tokens of the SDF graph with their time labels as a vector, then the execution of a single iteration of the graph reproduces the same set of initial tokens, but with new time labels and hence a new vector. This vector can be computed by multiplication with the $(\max, +)$-matrix, say $\mathbf{M}$. Hence, for an SDF graph, the evolution of its execution is captured by increasing powers of this matrix, $\mathbf{M}^k$. As in regular linear algebra, the long term behavior of linear systems is studied by means of *spectral analysis* techniques, i.e., computing eigenvalues and eigenvectors of the matrix. This is also true for $(\max, +)$-algebra and a rich literature exists on spectral analysis and algorithms to compute them. The most important result relevant for this section is that the largest eigenvalue corresponds to the worst-case average period with which the SDF graph execution will end. This is the reciprocal of the maximal throughput that is guaranteed to be achieved.

For SADF graphs, the situation is slightly more complicated. This section summarizes the results of [25], [26]. While an SDF graph executes the same behavior over and over again, an SADF will arbitrarily switch between behaviors. For an FSM-based SADF graph, the possible sequences of scenarios are specified by the language of the FSM and each individual scenario corresponds to an SDF iteration. This means that we can apply the same correspondence to associate with every scenario σ, a $(\max, +)$-matrix $\mathbf{M}_\sigma$. This means that the accumulated effect of a sequence $\sigma_1\sigma_2\ldots\sigma_n$ is captured by the product of the scenario matrices: $\mathbf{M}_{\sigma_n}\ldots\mathbf{M}_{\sigma_2}\mathbf{M}_{\sigma_1}$. We consider different options for analyzing this type of behavior. First, by constructing an explicit state space representation of all possible behaviors [25]. Second, by a transformation to a $(\max, +)$-automaton [25]. Third, by an approximation using invariants to separate scenario behaviors [26].

The explicit state space approach constructs the state space in the form of a labeled transition system. The states of the transition system capture states of an SADF at the start or end points of an execution of an iteration in a particular scenario. This state can be captured by two components. First, the state of the FSM; second, a vector of time stamps of the tokens in the graph. Because only the relative differences between time stamps are relevant for the future behavior of the graph, the vector is *normalized*, meaning that an identical constant is subtracted from all entries such that the maximum value in the vector becomes equal to 0. Edges in the state space are constructed as follows. From the starting FSM state all possible outgoing transitions are considered. For any such transitions, the corresponding scenario is executed, meaning that the vector of the starting state is multiplied by the corresponding matrix. The resulting vector is normalized to obtain together with the new FSM state, the new state in the state space. The constant subtracted for normalization is used, together with the scenario, as a label for the newly created edge. This label represents the amount of time passed in the specific scenario iteration.

For a self-timed bounded graph [13] with integer (or rational) execution times, the state space is finite. The main property of the state space is that the state of the SADF graph (worst-case state of the system) after a sequence of scenarios is found by following a path from the initial state through edges labeled with the appropriate scenarios. The final state is given by the normalized vector found in the final state, incremented with the sum of the delay labels along the path. It is not hard to see that for a finite state space, the worst-case throughput, measured in terms of the average number of iterations per time unit, is given by a path in the state space that has the largest average delay per edge. In other words, the maximum cycle mean (MCM) of the state space graph determines the throughput. The throughput measured in iterations per time unit can also be expressed as a throughput measured in actor firings per time unit since the number of firings of an actor per iteration is fixed.

If we want to explicitly construct the state space, we may find that in some cases, the state space can be large. There is a more efficient means for computing the worst-case throughput, by means of a conversion to a $(\max, +)$-automaton [25], [27]. This method intuitively relies on the observation that

dependencies that determine the time stamps of tokens are dependencies from individual tokens at the end of an iteration, to individual tokens at the beginning of the next iteration. Every pair of such tokens corresponds to a single entry in the scenario matrix. This means that the MCM in the state space is determined by a periodic set of dependencies between individual tokens. The $(\max, +)$-automaton we construct, can be seen as a directed multigraph which has a vertex for every combination of a state of the FSM and a single token in the graph in its initial state. Arcs in the graph are constructed from the edges in the scenario FSM. If the FSM has an edge from state q to q' labeled with scenario σ, then for every entry d, at row i, column j in the matrix $\mathbf{M}_\sigma$, we add an arc to the $(\max, +)$-automaton going from vertex (q, j), labeled d, to vertex (q', i). Now, the worst-case throughput can be determined as the reciprocal of the MCM in this graph. The size of this graph, in contrast to the explicit state space can be easily determined upfront. The number of vertices is equal to the number of states in the FSM multiplied by the number of initial tokens in the graph. The MCM computation on the $(\max, +)$-automaton is typically much faster than the MCM computation on the explicit state space [25].

The third approach, detailed in [26], tries to establish a conservative explicit state space that has a smaller size than the full state space. Intuitively, what the method does is to determine the normalized state vectors of the state space a-priori, by some heuristic method. Subsequently, the general state space construction method is followed, except that states will have, instead of the minimal state-vector obtained from the matrix multiplication, an a-priori defined reference vector. To make it a conservative bound, the delay annotation on an edge may need to be increased by a certain amount. This makes that the method may overestimate the average delay per edge and hence underestimate the achievable throughput. However, by limiting the vectors to pre-defined values, the size of the space-space can be limited. Heuristics for determining the invariant state vectors, can for instance allow only a single, universal vector, or may allow the state-vector to vary with the source and destination scenarios [28].

Stochastic throughput analysis. Worst-case analysis as discussed above gives a (conservative) bound on the minimal throughput. The stochastic variant of SADF also allows analysis of the typical throughput (or the typical throughput under worst-case execution time conditions) assuming self-timed (data-driven) execution. The stochastic equivalent to the guaranteed actor throughput is the long-run average number of firing completions of an actor per time unit, which is equal to the reciprocal of the long-run average latency between two firing completions of that actor. Computing such long-run averages is currently based on specialized extensions of techniques used in conventional model checking. These techniques are applicable to the generic form of SADF with scenario transitions during iterations and discrete execution time distributions. The approach relies on the semantics of SADF defined in [23] using *Timed Probabilistic Systems* (TPS) [29]. TPS extends a Markov decision process by explicitly distinguishing timeless action transitions from transitions for advancing time. The TPS underlying an SADF graph captures the non-determinism between performing enabled actions for different actors (non-determinism originates from concurrency only, in the context of this SADF variant [20]) as well as the probabilistic choices originating from the Markov chains associated with certain actors and the discrete execution time distributions. By shifting the information about the occurrence of actions or advancing time into the states of the TPS, a conventional Markov decision process is obtained for which temporal reward functions can be defined on the states to express any performance metric of interest (including throughput) [30]. After resolving non-determinism, a discrete Markov reward model is obtained where states contain information about the actions and advances in time that actually occurred. Computing the throughput therefore boils down to solving the Markov reward model, for which conventional techniques are applied [31].

An important step in the approach to stochastic throughput analysis is to resolve non-determinism. The policy used for resolving non-determinism may impact the result for certain performance metrics. An example is the maximum occupancy of a buffer, which depends on the order of scheduling the reading and writing of tokens. An important property of many dataflow models, including the stochastic variant of SADF, is however that time-dependent long-run averages are not affected by the policy used for resolving non-determinism [20]. The throughput is such a time-dependent long-run average.

The sketched approach is applicable to the generic form of SADF, however, a potential problem with this analysis approach is state space explosion. Two techniques are used to counter state space explosion substantially. We reduce the state space, using a technique of [31], to a discrete Markov reward model with only those states that are relevant for the performance metric of interest. The second approach to counter state space explosion involves an efficient on-the-fly construction of the reduced Markov reward model without the need to first construct the original TPS. This is possible because of specific semantic properties, such as persistence of enabled actions in bounded SADF graphs. The implementation in SDF^3 [32] uses a two-phase approach where the first phase constructs on-the-fly the reduced Markov chain while also computing the total amount of time passed between two firing completions of the considered actor. The second phase applies traditional techniques to solve this Markov reward model.

B. Latency Analysis

Worst-case latency analysis. For throughput analysis we have only looked at the average rate at which iterations of the graph are being executed. We did not care about the firing times of individual actor firings. In order to establish latency information we want to know at what time individual actor firings take place or complete within an iteration. Note that the explicit state space (either the exact one or the conservative approximation) contains sufficient information to determine such firing times (with the conservative state space leading to conservative estimates). It can be shown that the starting or completion time of any actor firing in some scenario can be expressed as a $(\max, +)$-linear combination of the entries in the state-vector defining the starting point of the iteration.

A common definition of latency is to derive linear bounds on actor firings of the form $t_k \leq \delta + \frac{k}{\tau}$ where t_k is the time

of the k-th firing, τ is the throughput, and δ a constant that can be interpreted as the latency of the actor. The latency can be determined by an exploration of the state space. The individual firings in the iterations can be determined from the starting state and δ can be determined from the maximal value of $t_k - \frac{k}{\tau}$ observed. Note that the exploration only needs to consider acyclic paths, because any cycle in the state space will not be faster than determined by the throughput and hence will not lead to a larger δ. It is interesting to note that besides determining the latency with respect to the maximum throughput, it is also possible to determine latency with respect to a given throughput requirement, by replacing τ in the calculation with the required throughput [26].

Stochastic latency analysis. Computing latency metrics is also supported for the generic variant of SADF with Markov chains associated to certain actors and discrete execution times. Such analysis relies on the same TPS-based approach as discussed for stochastic throughput analysis in Sec. III-A. In fact, computing throughput relies on computing the long-run average latency between successive firings of an actor. In addition to the long-run average inter-firing latency, it is also possible to compute the variance in this latency [33]. These metrics require probabilistic information to be available in the SADF graph. By discarding the probabilities, computing the minimum and maximum latency between successive firing completions of an actor is also supported using a similar TPS-based approach with temporal reward functions expressing the considered minimum or maximum. Besides these latency metrics, a similar TPS-based analysis approach allows computing finite horizon latency properties such as the minimum, maximum and expected response time [33]. The response time denotes the time of the first firing completion of an actor (for example the RC actor in Fig. 1 that reconstructs the video frame) and hence presents a very simple application of the TPS-based analysis approach. Finally, the stochastic SADF MoC enables analysis of the probability of a deadline miss (given as constraint for the response delay or inter-firing latency). Such analysis can for example be used in an MPEG-4 decoder to compute the probability that a frame is not processed before its deadline. This stochastic analysis technique is useful when designing soft real-time systems.

C. Buffer Analysis

Analyzing occupancy-related properties of the buffers incorporated in edges of an SADF graph is currently only supported with the TPS-based analysis approach. Although the main idea is the same as explained in Sec. III-A, a few additional aspects must be considered. First, an extended TPS semantics is needed compared to the one in [23]. This extended semantics ensures conservative results when computing the long-run time-weighted average buffer occupancy and long-run time-weighted variance in buffer occupancy. The crux is that reservation of buffer space at the beginning of and actor firing must be taken into account [31]. The reduced Markov reward model obtained for these metrics includes only those states in which space in a buffer is reserved and states in which buffer space is released. These actions depend on the behavior of respectively the producing and consuming actor. Note that these metrics require the time duration of each specific occupancy to be taken into account. As a result, analysis of the average and variance buffer occupancy metrics are the most computational intensive ones available for SADF. As indicated in Sec. III-A, we need to elaborate on resolving non-determinism when computing the maximum buffer occupancy. To ensure correct computation of the maximum buffer occupancy as defined here, the approach is to prioritize reserving buffer space over releasing buffer space.

During the design of a system one is often interested in dimensioning buffer sizes such that certain other performance criteria such as throughput are optimized [15]. Currently, there is only a TPS-based approach to compute the buffer occupancy that can occur maximally for any self-timed execution. Dimensioning buffer sizes larger than this maximum buffer occupancy will never be of practical use.

IV. Implementing SADF graphs

To implement an SADF graph, its actors and edges should be bound and scheduled on the resources of an MPSoC. Whenever multiple actors share a resource, an arbiter must schedule accesses to the resource. Due to the constant rates in a scenario of the SADF graph, all data dependencies between actors in this scenario are known at design time. This makes it possible to construct a static-order schedule for the actors that share a resource. Such a static-order schedule has almost no run-time overhead. It can be implemented with a sequence of function calls. A resource may also be shared between actors of different applications. Typically, the set of active applications is not known at design-time. Therefore it is not possible to construct a single static-order schedule for actors of different applications. In this situation, static-order schedules can be used for actors of the same application, but (actors of) different applications need to be handled by a run-time scheduler. Scheduling techniques such as time-division multiple-access (TDMA), priority-based budget scheduling (PBS) [34], and round-robin (RR) can be used. These scheduling strategies differ in their resource efficiency, predictability, and composability. Round-robin, for example, achieves a very good average processor utilization, but it has a high worst-case response time. It therefore has a low resource efficiency when timing guarantees need to be provided. TDMA and PBS are both predictable, i.e, they provide timing guarantees on their worst-case response times. TDMA is also composable, the timing behavior of one application cannot influence another application in any way. This composability comes, however, at the cost of a lower resource efficiency compared to PBS [34].

Each scenario in the SADF graph could in principle use a different mapping. To implement this, a run-time reconfiguration mechanism is needed that can transfer data items (tokens) and code (actors) between different memories whenever a scenario switch occurs. To provide timing guarantees, a design flow should take the overhead of the run-time reconfiguration into account. In the worst-case, a reconfiguration is performed after executing a single iteration of the graph. Hence, scenario switches can occur very frequently (the MPEG-4 decoder may switch scenarios 20 times per second). Providing timing guarantees while allowing such frequent reconfigurations may lead to large resource reservations. Therefore it is reasonable to assume that the actors of an SADF graph are mapped to the

same resources in all scenarios. This *unified mapping* avoids that data items or code need to be moved between different memories when switching between scenarios.

Stuijk et al. [35] present a design flow that maps a throughput-constrained application, modeled with an FSM-based SADF, to an MPSoC. The flow uses the technique from [15] to analyze the trade-off between storage space and throughput for each individual scenario in the graph. Although this technique can only analyze (C)SDF graphs, it can still be used, since an individual scenario in the SADF corresponds to an SDF graph. After computing the trade-off space for each individual scenario, the flow creates a unified trade-off space for all scenarios. This is done using Pareto algebra [36] by taking the free product of all trade-off spaces and selecting only the Pareto optimal points in the resulting space. A Pareto point with the smallest storage space that satisfies the throughput constraint is then used to constrain the storage requirements of the edges in the SADF graph. Note that an analysis technique that works directly on SADF graphs may be able to find a smaller storage space assignment that satisfies the same throughput constraint. Development of such an analysis technique is still an open problem. After constraining the storage space of the edges, the flow performs a unified binding of the actors to the MPSoC resources. Next, static-order schedules are constructed for all processors to which actors have been bound. Finally, the flow computes the minimal TDMA time slices needed on these processors to guarantee that the throughput constraint of the application is met. By minimizing the TDMA time slices, processor resources are saved for other applications. The output of the flow is a set of Pareto optimal mappings that provide a trade-off in their resource usage. In some of these mappings, the application could for example use a lot of computational resources, but limited storage resources, whereas an opposite situation may be obtained in other mappings. At run-time the most suitable mapping can then be selected based on the resource usage of the applications which are already running on the platform [37], [38].

The SDF^3 tool set [32] implements the design flow outlined above. In [35], it is used to map an MPEG-4 video decoder and an MP3 audio decoder onto a three processor MPSoC. The experimental results show that the design flow is able to achieve substantial resource savings compared to an SDF-based approach. A saving of 66% in processor resources is obtained for the MPEG-4 decoder. For the MP3 decoder, the memory and interconnect bandwidth usage are reduced by respectively 21% and 23%. This demonstrates the potential offered by the SADF MoC to exploit the dynamic behavior inside applications in order to save resources.

V. Comparison with other Dataflow Models-of-Computation

Many different dataflow Models of Computations (MoCs) have been developed. These MoCs differ in their expressiveness and succinctness, analyzability and implementation efficiency. The *expressiveness* and *succinctness* of a model indicate which systems can be modeled and how compact these models are. For example, the behavior of an MPEG-4 SP decoder can be modeled with an SDF graph consisting of 4 actors. The same behavior can be modeled with an HSDF graph containing 201 actors. Hence, the SDF graph is more succinct than the HSDF graph. MoCs can also be compared on their analyzability. The *analyzability* of a MoC is determined by the availability of analysis and synthesis algorithms and the run-time needed for an algorithm on a graph with a given number of nodes. The third aspect that is relevant when comparing MoCs is their *implementation efficiency*. This is influenced by the complexity of the on-line or run-time scheduling problem and the (code) size of the resulting schedules. Also for this aspect it is important that different MoCs are compared using graphs with an equal number of nodes. This decouples the succinctness from the other aspect that are considered. This section compares the most important dataflow MoCs on the three aforementioned aspects. Fig. 2 visualizes the result of this comparison. The left side of Fig. 2 shows an expressiveness hierarchy for these MoCs. An edge is drawn from MoC A to MoC B if an instance of MoC A is also an instance of MoC B, or if a transformation is known that can transform a model of MoC A to a semantically equivalent model of MoC B. We only show relations that are (formally or informally) reported in the literature and those that are straightforward. The details of semantic aspects preserved by the individual relations may differ. In some cases, for example, auto-concurrent actor firings need to be excluded. In other cases, only untimed behavior is considered. Some models are incomparable and therefore placed in separate branches. Consider for example the Cyclo-Static Dataflow (CSDF) MoC [39], [40], which allows the rate of a port to change between subsequent firings. The sequence of the rates of the port must be finite and periodically repeating. It allows port rates to be equal to zero. In contrast, the FSM-based SADF model does not require a periodically repeating sequence, but it requires all port rates to be positive. Hence, these models can express different systems and are therefore on separate branches in Fig. 2. A more precise and complete investigation of the expressiveness hierarchy is an interesting subject for future work, but out of the scope of the current paper. The right part of Fig. 2 compares all MoCs in the hierarchy on the three mentioned aspects of expressiveness, analyzability and implementation efficiency. It is our personal assessment, because it is difficult, if not impossible, to formalize this comparison. The expressiveness and succinctness axis orders all MoCs in terms of their ability to capture dynamic behavior in a compact way. The order is consistent with the hierarchy in the left part of the figure.

There exists a range of dataflow MoCs that cannot express dynamic applications. MoCs such as (homogeneous) synchronous dataflow [10], computation graphs (CG) [41], marked graphs (MG) [42], and weighted marked graphs (WMG) [43] assume that actors have a fixed production and consumption rate on each firing. Note that HSDF and SDF are identical to the Petri-net subclasses MG and WMG, respectively. HSDF/MG assumes that all rates are equal. Many analysis algorithms for these MoCs have polynomial complexity and efficient implementations can be derived. The other MoCs support multi-rate dependencies. This makes these MoCs more succinct, but it also implies that design-time

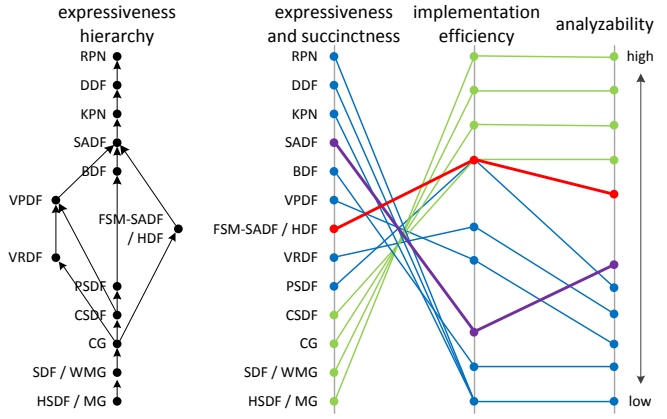

Fig. 2. Comparison of dataflow models of computation.

analysis and run-time scheduling of these MoCs is more complex. Computation graph actors may consume a different number of tokens than the number of tokens needed to enable the firing of the actor. This makes analysis and implementation of this MoC even more complex than the SDF/WMG MoC. In [39] it is shown that the CSDF MoC is as expressive as the HSDF MoC, but it is more compact for certain aspects. The changing port rates make analysis and scheduling of CSDF more complex compared to SDF.

There exist several dataflow MoCs that can incorporate some dynamism, allow design-time analysis, and can be implemented reasonably efficiently. In earlier sections, FSM-based SADF/HDF has already been discussed extensively. FSM-based SADF offers many analysis algorithms, which typically have a run-time similar to analysis algorithms for CSDF. Sec. IV shows that it is further possible to derive efficient implementations for FSM-based SADF. The implementation efficiency is also similar to that for CSDF. Parameterized Synchronous Dataflow (PSDF) [44] allows the rates of one or more ports to be parameterized rather than constant. Parameterized schedules and buffer sizes can be computed. Options to express dynamism are however limited. Variable Rate Dataflow (VRDF) [45] does not require constant rates. Port rates are allowed to vary arbitrarily within a specified range. Variable Phased Dataflow (VPDF) [46] is a CSDF-like extension of VRDF where the number of repetitions of CSDF phases can be parameters from some finite interval. Existing analyses of VRDF and VPDF are limited to computing (conservative) buffer sizes under a throughput constraint.

Dataflow MoCs such as Boolean Dataflow (BDF) and Dynamic Dataflow [47] allow data-dependent firing rules. This makes them Turing-complete in the sense that they can operationally simulate a Turing machine. Consequently, it is impossible to realize an exact analysis of their timing behavior and buffer sizes at design-time. These MoCs require run-time scheduling and deadlock detection. This makes their implementation far less efficient compared to all MoCs discussed so far. The Kahn Process Network (KPN) [17] is another MoC that can be used to express application dynamism. The Reactive Process Network (RPN) [48] MoC extends KPN with state transitions that allow it to change the function of the process network based on events. Both KPN and RPN do not allow for design-time analysis and require a complex run-time mechanism that incurs a large implementation overhead. We consider DDF more expressive than KPN because the (informal) definition of DDF given in [47] allows non-functional (indeterminate) behavior, which cannot be expressed in KPN.

Finally, we consider the general SADF MoC. FSM-based SADF/HDF allows the graph structure to change with each iteration. General SADF allows it also to change inside an iteration. The BDF MoC also allows changing rates inside an iteration, but it only allows a limited set of constructs. This makes the general SADF MoC more expressive than BDF. However, the general SADF MoC does not allow arbitrary rates. Therefore, it is placed below the KPN MoC on the expressiveness axis. Sec. III shows that a broad class of analysis techniques is available for the SADF MoC. These may suffer from state space explosion problems, however, making this MoC less analyzable. Because it is possible to change scenarios inside an iteration, the general SADF MoC requires run-time scheduling. This makes its implementation not very efficient.

An overall conclusion is that expressiveness is typically traded off against analyzability and implementation efficiency. Because of its ability to express dynamism while allowing design-time analysis and efficient implementation, the FSM-based SADF MoC provides an interesting trade-off between expressiveness, analyzability, and implementation efficiency.

VI. SDF^3 Tool Set

The open-source SDF^3 tool set [32] implements all analysis and implementation techniques discussed in this paper. The tool set also offers an SADF graph generation algorithm that constructs random SADF graphs which are connected, consistent, and deadlock-free. This generation algorithm can be used to benchmark novel SADF analysis, transformation, and implementation algorithms. If desired, the user can restrict relevant properties of the generated graph (e.g., limit port rates, or construct only acyclic or strongly connected graphs).

All algorithms and techniques implemented in SDF^3 can be accessed through a set of command line tools as well as a C/C++ API. Besides the SADF MoC, the tool set offers analysis, transformation, generation, and implementation techniques for the SDF and CSDF MoCs. Algorithms are provided to transform dataflow graphs from one MoC to (conservative) dataflow graphs in another MoC. The rich set of algorithms offered by SDF^3, makes it a versatile tool set for the development of novel dataflow-based design approaches.

VII. Conclusions

Embedded systems nowadays typically run multiple applications, such as multimedia and wireless, concurrently on a heterogeneous MPSoC. Model-based design approaches are used to map these timing-constrained applications to the MPSoC. The Model-of-Computation used by these approaches may differ in its design-time analyzability, expressiveness, and implementation efficiency. Many dataflow-oriented MoCs have been proposed in the past. In this paper, we compare different dataflow models on the aforementioned aspects. This comparison shows that many MoCs that allow design-time

analysis and an efficient implementation are not able to capture the dynamic behavior of applications. On the other hand, modern streaming applications exhibit dynamic behavior. This dynamism should be captured in order to reduce the resource usage of the system. The Scenario-Aware Dataflow (SADF) MoC addresses this challenge. SADF is able to capture the dynamic behavior of applications while offering a large set of analysis techniques and efficient implementations. Two challenging problems remain. First, it would be interesting to further investigate and formalize the relations between the different dataflow MoCs discussed in this paper. Second, novel techniques are needed to model implementation decisions directly into SADF graphs and/or to take them into account in the analysis. Computation of buffer sizes directly on SADF graphs is for example still an open problem. Such techniques could potentially lead to resource savings in implementations.

REFERENCES

[1] O. Gangwal *et al.*, *Dynamic and Robust Streaming In and Between Connected Consumer-Electronics Devices*. Springer, 2005, vol. 3, ch. Building Predictable Systems on Chip: An Analysis of Guaranteed Communication in the AEthereal Network on Chip, pp. 1–36.

[2] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design and Test of Computers*, vol. 18, no. 6, pp. 23–33, 2001.

[3] G. Martin, "Overview of the MPSoC design challenge," in *Design Automation Conf., DAC 06, Proc.* ACM, 2006, pp. 274–279.

[4] A. Bonfietti *et al.*, "An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms," in *Int. Conf. on Design, Automation and Test in Europe, DATE 10, Proc.*, 2010.

[5] W. Haid *et al.*, "Multiprocessor SoC software design flows," *Sig. Proc. Mag.*, vol. 26, no. 6, pp. 64 –71, 2009.

[6] W. Liu *et al.*, "Efficient SAT-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization," in *Real-Time Systems Symp., RTSS 08, Proc.* IEEE, 2008, pp. 492–504.

[7] O. Moreira *et al.*, "Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix," in *Real Time and Embedded Technology and Applications Symp., Proc.* IEEE, 2005, pp. 332–341.

[8] A. Pimentel, "The artemis workbench for system-level performance evaluation of embedded systems," *Int. J. of Embedded Systems*, vol. 3, no. 3, pp. 181–196, 2008.

[9] S. Stuijk *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Design Automation Conf., Proc.* ACM, 2007, pp. 777–782.

[10] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[11] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. CRC Press, 2009.

[12] A. Ghamarian *et al.*, "Throughput analysis of synchronous data flow graphs," in *Int. Conf. on Application of Concurrency to System Design, ACSD 06, Proc.* IEEE, 2006, pp. 25–36.

[13] A. Ghamarian *et al.*, "Liveness and boundedness of synchronous data flow graphs," in *Int. Conf. on Formal Methods in Computer Aided Design, FMCAD 06, Proc.* IEEE, 2006, pp. 68–75.

[14] A. Ghamarian *et al.*, "Latency minimization for synchronous data flow graphs," in *Conf. on Digital System Design, DSD 07, Proc.* IEEE, 2007, pp. 189–196.

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

[16] M. Wiggers *et al.*, "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure," in *Int. Conf. on Hardware-Software Codesign and System Synthesis, CODES+ISSS 06, Proc.* ACM, 2006, pp. 10–15.

[17] G. Kahn, "The semantics of a simple language for parallel programming," in *Inf. Proc., IFIP 74, Proc.* North-Holland, 1974, pp. 471–475.

[18] T. Parks, "Bounded Scheduling of Process Networks," Ph.D. dissertation, University of California, EECS Dept., Berkeley, CA, 1995.

[19] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks," in *European Symp. on Programming, ESOP 03, Proc., LNCS 2618*. Springer, 2003, pp. 319–334.

[20] B. Theelen *et al.*, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Int. Conf. on Formal Methods and Models for Co-Design, MEMOCODE, Proc.* IEEE, 2006, pp. 185–194.

[21] S. Gheorghita *et al.*, "System-scenario-based design of dynamic embedded systems," *ACM Trans. on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1–45, 2009.

[22] A. Girault, B. Lee, and E. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 742–760, 1999.

[23] B. Theelen *et al.*, "Scenario-aware dataflow," TU Eindhoven, Tech. Rep. ESR-2008-08, 2008.

[24] F. Baccelli *et al.*, *Synchronization and linearity: an algebra for discrete event systems*. Wiley, 1992. [Online]. Available: http://www-rocq.inria.fr/metalau/cohen/SED/book-online.html

[25] M. Geilen and S. Stuijk, "Worst-case performance analysis of synchronous dataflow scenarios," in *Int. Conf. on Hardware-Software Codesign and System Synth., CODES+ISSS, Proc*. ACM, 2010, pp. 125–134.

[26] M. Geilen, "Synchronous dataflow scenarios," *ACM Trans. Embedded Computing Systems*, vol. 10, no. 2, pp. 16:1–16:31, 2010.

[27] S. Gaubert, "Performance evaluation of (max, +) automata," *IEEE Trans. on Automatic Control*, vol. 40, no. 12, pp. 2014–2025, 1995.

[28] P. Poplavko, M. Geilen, and T. Basten, "Predicting the throughput of multiprocessor applications under dynamic workload," in *Int. Conf. on Computer Design, ICCD 10, Proc.* IEEE, 2010, pp. 282–288.

[29] L. de Alfaro, "Formal verification of probabilistic systems," Ph.D. dissertation, Stanford University, 1997.

[30] J. Voeten, "Performance evaluation with temporal rewards," *Performance Evaluation*, vol. 50, no. 2–3, pp. 189–218, 2002.

[31] B. Theelen, "Performance modelling for system-level design," Ph.D. dissertation, TU Eindhoven, 2004.

[32] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Int. Conf. on Application of Concurrency to System Design, ACSD 06, Proc.* IEEE, 2006, pp. 276–278.

[33] B. Theelen, "A performance analysis tool for scenario-aware streaming applications," in *Int. Conf. on Quantitative Evaluation of Systems, Proc.* IEEE, 2007, pp. 269–270.

[34] M. Steine, M. Bekooij, and M. Wiggers, "A priority-based budget scheduler with conservative dataflow model," in *Conf. on Digital System Design, DSD 09, Proc.* IEEE, 2009, pp. 37–44.

[35] S. Stuijk, M. Geilen, and T. Basten, "A predictable multiprocessor design flow for streaming applications with dynamic behaviour," in *Conf. on Digital System Design, DSD 10, Proc.* IEEE, 2010, pp. 548–555.

[36] M. Geilen *et al.*, "An algebra of pareto points," *Fundamenta Informaticae*, vol. 78, no. 1, pp. 35–74, 2007.

[37] H. Shojaei *et al.*, "A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management," in *Design Automation Conf., DAC 09, Proc.* ACM, 2009, pp. 917–922.

[38] C. Ykman-Couvreur *et al.*, "Fast multi-dimension multi-choice knapsack heuristic for MP-SoC run-time management," in *Int. Symp. on SoC, Proc.* IEEE, 2006, pp. 1–4.

[39] R. Lauwereins *et al.*, "Geometric parallelism and cyclo-static data flow in GRAPE-II," in *Int. Work. on Rapid System Prototyping, Proc.* IEEE, 1994, pp. 90–107.

[40] G. Bilsen *et al.*, "Cyclo-static dataflow," *IEEE Trans. on Sig. Proc.*, vol. 44, no. 2, pp. 397–408, 1996.

[41] R. Karp and R. Miller, "Properties of a model for parallel computations: Determinancy, termination, queueing," *SIAM J. of Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[42] F. Commoner *et al.*, "Marked directed graphs," *J. of Comp. and Sys. Sci.*, vol. 5, no. 5, pp. 511–523, 1971.

[43] E. Teruel *et al.*, "On weighted T-systems," in *Int. Conf. on Application and Theory of Petri Nets, ATPN 92, Proc.* Springer, 1992, pp. 348–367.

[44] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Trans. on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.

[45] M. Wiggers, M. Bekooij, and G. Smit, "Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication," in *Real-Time and Embedded Technology and Applications Symp., RTAS 08, Proc.* IEEE, 2008, pp. 183–194.

[46] M. Wiggers, M. Bekooij, and G. Smit, "Buffer capacity computation for throughput-constrained modal task graphs," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 1–59, 2010.

[47] J. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, UC Berkeley, 1993.

[48] M. Geilen and T. Basten, "Reactive process networks," in *Int. Conf. on Embedded Software, EMSOFT 04, Proc.* ACM, 2004, pp. 137–146.