Model Checking of Finite-state Machine-based Scenario-aware Dataflow Using Timed Automata

Mladen Skelin
Department of Engineering Cybernetics,
Norwegian University of Science and Technology
mladen.skelin@itk.ntnu.no

Erik Ramsgaard Wognsen, Mads Chr. Olesen,
René Rydhof Hansen, Kim Guldstrand Larsen
Department of Computer Science, Aalborg University
{erw,mchro,rrh,kgl}@cs.aau.dk

***Abstract*—Dataflow formalisms are widely used for modeling and analyzing streaming applications. An important distinction is between static and dynamic formalisms, the latter allowing for the workload to change on-the-fly. The recently introduced finite-state machine based scenario aware dataflow (FSM-SADF) is a dynamic dataflow formalism that increases the expressiveness of the static synchronous dataflow (SDF) formalism, by allowing finite-state control, while to a large extent retaining its design-time analyzability.**

This paper reports on the translation of the FSM-SADF formalism to UPPAAL timed automata that enables a more general verification than currently supported by existing tools. We base our translation on a compositional approach where the input FSM-SADF model is represented as a parallel composition of its integral components modeled as automata. Thereafter, we show how to model check quantitative and qualitative properties both supported and not supported by the existing tools. We demonstrate our approach on a realistic case study from the multimedia domain.

I. Introduction

Thanks to their relatively simple graphical representation, compactness and the ability to easily express parallelism, dataflow formalisms have become almost irreplaceable tools for the design and analysis of (embedded) signal processing and streaming applications. Dataflow formalisms are instantiated as directed graphs where vertices are called *actors* and edges are called *channels*. Actors represent application tasks, while channels represent data dependencies between actors. In dataflow, an actor *firing* is an indivisible quantum of computation during which an actor consumes a certain number of data values from its input channels and produces a certain number of data values on its output channels. These data values are abstracted into *tokens*, while the number of tokens consumed/produced are called *rates*. In timed dataflow formalisms, it takes some time before the actor firing completes. This time duration is called the *actor firing duration*.

Modern signal processing and streaming applications exhibit dynamic behaviour, i.e., their workload changes over time to, e.g., accommodate different multimedia frame types. We call such applications dynamic applications. Depending on their ability to capture dynamic applications, dataflow formalisms can be divided into two groups: static and dynamic [5] ones. Static dataflow formalism cannot model dynamic applications, while the dynamic ones can.

Static dataflow formalisms such as synchronous dataflow (SDF) [12] are characterized by their compile time predictability (analyzability), implementation efficiency (low run-time overhead) and high optimization potential at the price of reduced expressiveness [17]. Dynamic dataflow formalisms on the other hand, achieve a higher level of expressiveness by sacrificing analyzabilty and implementation efficiency. Increased expressiveness even renders some formalisms undecidable. An example of such a formalism is the Turing complete boolean dataflow (BDF) [6]. A comparison of the most prominent dataflow formalisms in terms of expressiveness, analyzability and implementation efficiency can be found in [17].

The recently introduced scenario-aware dataflow (SADF) formalism [20] captures application dynamism using *scenarios*. Scenarios represent distinct application operating modes that occur during its lifetime. Rates and actor firing durations differ from one scenario to the other. The data dependent conditions that determine scenario occurrence patterns are abstracted into Markov chains. SADF to a large extent preserves SDF's compile-time analyzability [17]. State-of-the-art SADF techniques are implemented in the SDF^3 tool [16].

Finite-state machine-based SADF (FSM-SADF) [9], [18] is a restricted form of SADF introduced to speed-up the analysis of the original formalism. FSM-SADF has been used in several important modeling [14], [15] and optimization contexts [7]. It is a restricted form because unlike SADF, it does not support hierarchical control through the use of nested Markov chains. This means that FSM-SADF cannot support sub-scenarios, i.e. in FSM-SADF actor rates and firing durations can change only at scenario boundaries while in SADF, they can change even within a scenario. In addition, FSM-SADF uses fixed actor firing durations per scenario, while SADF uses discrete distributions per scenario. Furthermore, unlike the probabilistic abstraction approach of SADF, FSM-SADF uses a non-deterministic abstraction where scenario sequences are specified by a non-deterministic FSM. These restrictions render the FSM-SADF analysis faster than the analysis of SADF, i.e. FSM-SADF is more analyzable than SADF. Also, FSM-SADF outcompetes SADF in terms of implementation efficiency [17]. However, the analysis might be less precise due to the abstractions made. On the other hand, FSM-SADF extends SADF by allowing actor-level auto-concurrency (simultaneous executions of a particular actor) which is explicitly prohibited in SADF as it may violate the determinacy of the model [20]. Thus, the parallelism embedded in an FSM-SADF specification is implicitly greater than the one embedded in an SADF specification. State-of-the art FSM-SADF techniques are implemented in the SDF^3 tool [16].

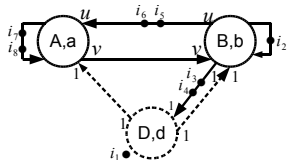
However, tools such as SDF^3 can be too specialized in the sense that they can only handle predefined properties, thus lacking support for analyzing user-defined properties. To circumvent this limitation, in this paper we propose a translation of the FSM-SADF formalism to timed automata (TA) [3] as the first step to enable more general verification. This is our first contribution. Using TA has a number of advantages, in that very efficient abstractions exist. For example, temporal logics can express many of the properties common in reasoning about timed systems with concurrency. Furthermore, TA models of dataflow specifications can be easily extended to add costs such as energy, and include the behaviour of the underlying implementation platform. This would in the future give us the possibility of using FSM-SADF for reachability analysis of embedded dynamic streaming applications through an optimal control formulation using model-checking techniques. Although other members of the SADF MoC family have been translated to model checkers before [11], [21], [22] our translation is (to the best of our knowledge) the first one that allows auto-concurrency in the model and is able to assure determinacy in its presence, by using the scenario-level FIFO policy of [9] in a "model checking" context. This is our second contribution. We demonstrate our approach using a multimedia case study modeled as an FSM-SADF graph (FSM-SADFG) for which we compute important quantitative and qualitative properties, some of which are not supported by the SDF^3 tool. We use the UPPAAL [4] state-of-the-art TA model checker.

II. Related Work

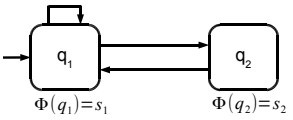
The SADF model has already been subjected to model checking in [21], which discusses a performance model-checking approach for SADF where its semantics is based on a timed probabilistic (labeled transition) system (TPS). In the TPS of SADF non-determinism can be arbitrarily resolved as it originates from concurrency of the actors, i.e. the ordering of timeless actions will not affect the overall behaviour of the system. We call this property of SADF *action determinacy* (or the diamond-property in a non-probabilistic setting). This property is the key to efficient analysis implemented in SDF^3 [16]. However, SDF^3 is not a general model-checker and it supports only a set of predefined properties such as deadlock freedom, maximum buffer occupancy, inter-firing latency, etc.

Theelen et al. [22] report on the use of the Construction and Analysis of Distributed Processes (CADP) tool suite in model checking of SADF using interactive Markov chains (IMC). The inability of IMC in supporting probabilistic choices is compensated by CADP. Also, IMC relies on exponentially distributed time, and therefore the original discrete distribution of SADF needed to be replaced by a single exponential distribution. Therefore, CADP may not always deliver the same result as SDF^3. Moreover, CADP is unable to evaluate reward-based properties and therefore cannot be used for the computation of throughput and buffer occupancy.

The work of Katoen et al. [11] extends the framework of [22] by introducing Markov automata (MA) based semantics of SADF. MA is a combination of probabilistic automata and Markov decision processes. The firing durations of actors are specified by negative exponential distributions. State-space reductions are based on confluence reduction which utilizes action determinacy for SADF actors. The approach can be used to obtain quantitative properties such as buffer occupancy, latency and throughput.



(a) Scenario graph



(b) Scenario FSM

	u	v
Scenario s_1	1	1
Scenario s_2	1	2

(c) Rates in scenarios s_1 and s_2

	a	b	d
Scenario s_1	4	1	1
Scenario s_2	2	5	1

(d) Actor firing durations in scenarios s_1 and s_2

Fig. 1: Example FSM-SADF graph

What is common to the aforementioned approaches is that they all consider SADF. In FSM-SADF on the other hand, non-determinism is an explicit property of the model, and not a side-effect of concurrency. In addition, FSM-SADF allows actor-level auto-concurrency which is explicitly prohibited in SADF. Geilen et al. [9] introduces the *(max,+)* algebraic semantics of FSM-SADF that can be used to obtain worst-case throughput and latency values in the presence of auto-concurrency. However, due to the nature of the *(max,+)* representation of a scenario, the approach of [9] can only give insight into the model's temporal behaviour at scenario boundaries. Therefore, the analysis of [9] is limited to throughput and latency computations only.

Fakih et al. [8] and Ahmad et al. [1] have previously used TA to model SDF, the original synchronous dataflow formalism. However, the TA model of SDF cannot be used to capture FSM-SADF as FSM-SADF unlike SDF includes a combination of streaming data and control. Moreover, the works of [1], [8] are more concerned with modelling lower-level details of the scheduling on a given execution platform.

III. Definition of FSM-SADF

Before we formally define the FSM-SADF formalism, we exemplify using the FSM-SADFG of Fig. 1. In FSM-SADF, two types of processes can be distinguished: *kernels* and *detectors*. Process is another name for an actor and we use the terms interchangeably. Kernels specify the data processing part of the application, while detectors model the control part of the application. The scenario graph of Fig. 1a consists of three vertices representing processes. Processes A and B are kernels (continuous lines) while process D (dashed line) serves as a detector. In FSM-SADF there can only exist one detector, i.e. the control is global. D determines in which scenario kernels A and B operate by sending them *control tokens* via control channels (dashed lines). Control tokens are valued. Tokens exchanged over data channels (continuous lines) are called *data tokens* and we abstract from their values. Channels are considered as FIFO buffers of infinite capacity. We use the terms channel and buffer interchangeably. The graph has 8 initial tokens labeled $i_1, \ldots, i_8$. Token i_1 is the initial token of the detector's implicit self-edge (channel with the same source and destination actor). This edge is usually not drawn as

FSM-SADF prohibits auto-concurrency for the detector. The running example defines two scenarios: s_1 and s_2. Depending on the operating scenario, the graph will change its properties. Fig. 1c specifies changes for graph rates over scenarios, while Fig. 1d specifies how process firing durations change over scenarios. The running example defines the FSM of Fig. 1b that determines the possible scenario occurrences where every FSM state is labeled with a scenario: state q_1 is labeled with s_1 and q_2 is labeled with s_2.

Our full definition of FSM-SADF follows here. Compared with [18], it is more concise because it is not necessary to represent sets of detectors and ports explicitly.

Definition 1 (FSM-SADF graph)**.** *An FSM-SADF graph is a tuple $G = (\mathcal{S}, \mathcal{K}, \mathcal{B}, E, R_p, R_c, \mathbb{S}, \mathbb{T}, \iota, \Phi, t, \phi_\iota, \psi_\iota)$, where*

1) *$\mathcal{S}$ is the nonempty finite set of scenarios,*
2) *$\mathcal{K}$ is the nonempty finite set of kernels,*
   - *$\mathcal{P} = \mathcal{K} \cup \{d\}$, where $d \notin \mathcal{K}$ denotes the unique detector, is the set of processes,*
3) *$\mathcal{B} \subseteq \mathcal{K} \times \mathcal{P}$ is the set of buffers,*
4) *$E : \mathcal{P} \times \mathcal{S} \to \mathbb{N}_0$ is the execution time for each process in each scenario,*
5) *$R_p, R_c : \mathcal{B} \times \mathcal{S} \to \mathbb{N}_0$ is the production (consumption) rate of the kernel producing to (process consuming from) each buffer in each scenario,*
6) *$(\mathbb{S}, \mathbb{T}, \iota, \Phi)$ is the FSM of the detector, where $\mathbb{S}$ is the nonempty set of states, $\mathbb{T} : \mathbb{S} \to 2^{\mathbb{S}}$ is the transition function, $\iota \in \mathbb{S}$ is the initial state, and $\Phi : \mathbb{S} \to \mathcal{S}$ associates each state with a scenario,*
7) *$t : \mathcal{K} \times \mathcal{S} \to \mathcal{S}^+$ is the string of scenarios sent to the FIFO of each kernel in each scenario of the detector,*
8) *$\phi_\iota : \mathcal{B} \to \mathbb{N}_0$ is the initial buffer status,*
9) *$\psi_\iota : \mathcal{K} \to \mathcal{S}^*$ is the initial control status.*

The detector is connected to every kernel by an explicitly ordered (FIFO) control channel. We further define $In(p) = \{b \in \mathcal{B} \mid \pi_r(b) = p\}$, where π_r is the right projection function, to be the set of buffers that process p consumes from (that input into p). Similarly, $Out(k) = \{b \in \mathcal{B} \mid \pi_l(b) = k\}$.

In anticipation of the next section we define $\varnothing$ to be the empty multiset, $\mathbb{P}$ to be the set of all submultisets of its input set, $\uplus$ to be the multiset sum, and $\setminus$ to be the zero-truncated asymmetric multiset difference. For example let $A = \{1, 1\}$ and $B = \{1, 2\}$. Then $A \cup B = \{1, 1, 2\}$ (maxima of multiplicities), $A \uplus B = \{1, 1, 1, 2\}$ (sums of multiplicities), $A \setminus B = \{1\}$, and $B \setminus A = \{2\}$. For strings $\sigma, \tau, \nu \in \mathcal{S}^*$ we define σ_i to be the ith element of σ, $\sigma + \tau$ to be the concatenation of σ and τ, and, if $\nu = \sigma + \tau$, then $\nu - \sigma = \tau$.

A. Operational Semantics

The behavior of an FSM-SADF graph is defined as a transition system where states are configurations.

Definition 2 (Configuration)**.** *A configuration of an FSM-SADF graph $G = (\mathcal{S}, \mathcal{K}, \mathcal{B}, E, R_p, R_c, \mathbb{S}, \mathbb{T}, \iota, \Phi, t, \phi_\iota, \psi_\iota)$ is a tuple $(\phi, \psi, \kappa, \delta)$, where ϕ is a buffer status, ψ a control status, κ a kernel status, and δ a detector status:*

- *A buffer status is a function $\phi : \mathcal{B} \to \mathbb{N}_0$ from each buffer to the number of tokens it stores,*
- *A control status is a function $\psi : \mathcal{K} \to \mathcal{S}^*$ from each kernel to the string of scenarios (control tokens) its FIFO stores,*
- *A kernel status is a function $\kappa : \mathcal{K} \to \mathbb{P}(\mathcal{S} \times \mathbb{N}_0)$ that to each kernel assigns a multiset of ongoing firings and their remaining execution times,*
- *A detector status is a pair $\delta \in \mathbb{S} \times (\mathbb{N}_0 \cup \{-\})$ that represents the state of the FSM and the remaining execution time of the ongoing firing, or, if there is no ongoing firing, the value $-$.*

The initial configuration of G is $(\phi_\iota, \psi_\iota, \kappa_\iota, \delta_\iota)$, where ϕ_ι and ψ_ι are defined in G, $\kappa_\iota = \mathcal{K} \times \{\varnothing\}$ and $\delta_\iota = (\iota, -)$.

Five types of configuration transitions are distinguished.

Definition 3 (Kernel Start Action)**.** *A kernel start action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{start}(k)} (\phi', \psi', \kappa', \delta)$ represents the start of a firing of kernel k. Let $s = \psi(k)_1$ denote the scenario of the firing (if it is defined). The transition is enabled if $|\psi(k)| \geq 1$ and $\forall b \in In(k) : \phi(b) \geq R_c(b, s)$. The resulting statuses are defined as*

$$\begin{aligned}
&\phi' = \phi[b \mapsto \phi(b) - R_c(b, s)] \quad \textit{for all } b \in In(k) \\
&\psi' = \psi[k \mapsto \psi(k) - s] \\
&\kappa' = \kappa[k \mapsto \kappa(k) \uplus \{(s, E(k, s))\}]
\end{aligned}$$

Definition 4 (Kernel End Action)**.** *A kernel end action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{end}(k)} (\phi', \psi, \kappa', \delta)$ is the end of a firing of kernel k. It is enabled if $\exists s \in \mathcal{S} : (s, 0) \in \kappa(k)$. The resulting buffer and kernel statuses are*

$$\begin{aligned}
&\phi' = \phi[b \mapsto \phi(b) + R_p(b, s)] \quad \textit{for all } b \in Out(k) \\
&\kappa' = \kappa[k \mapsto \kappa(k) \setminus \{(s, 0)\}]
\end{aligned}$$

Definition 5 (Detector Start Action)**.** *A detector start action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{start}(d)} (\phi', \psi, \kappa, \delta')$ represents the start of a firing of the detector, d. It is enabled if there is no ongoing firing $\exists s \in \mathbb{S} : \delta = (s, -)$ and all inputs are available $\forall b \in In(d) : \phi(b) \geq R_c(b, \Phi(s))$. The resulting statuses are*

$$\begin{aligned}
&\phi' = \phi[b \mapsto \phi(b) - R_c(b, \Phi(s))] \quad \textit{for all } b \in In(d) \\
&\delta' = (s, E(d, \Phi(s)))
\end{aligned}$$

Definition 6 (Detector End Action)**.** *A detector end action transition $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{end}(d)} (\phi, \psi', \kappa, \delta')$ is enabled if $\exists s \in \mathbb{S} : \delta = (s, 0)$, and the resulting statuses are*

$$\begin{aligned}
&\psi' = \psi[k \mapsto \psi(k) + t(k, \Phi(s))] \quad \textit{for all } k \in \mathcal{K} \\
&\delta' = (s', -) \quad \textit{for some } s' \in \mathbb{T}(s)
\end{aligned}$$

In [18] time transitions are defined very generally, such that to account for given scheduling/resource constraints one needs to instantiate the time transitions needed. In the following we will assume a unconstrained execution, namely that all ongoing firings advance at the same pace.

Definition 7 (Time Transition)**.** *A time transition* $(\phi, \psi, \kappa, \delta) \xrightarrow{\text{time}(t)} (\phi, \psi, \kappa', \delta')$ *represents time progressing* t *time units. It is enabled if no kernel end or detector end transition is enabled, and* t *is the smallest remaining execution time of any ongoing firing. The resulting kernel status is*

$$\kappa' = \kappa[k \mapsto \{(s, n-t) | (s, n) \in \kappa(k)\}] \quad \textit{for all } k \in \mathcal{K}$$

using multiset comprehension. The detector status $\delta = (s, n)$ *is updated as* $\delta' = (s, n-t)$*, unless* $n = -$ *in which case it is unchanged,* $\delta' = \delta$*.*

B. Overtaking Problem and Determinacy

The operational semantics of Section III-A allows simultaneous executions of a particular kernel, i.e. auto-concurrency. Note that the kernel status of Definition 2 entails a multiset of ongoing firings. In the case of the detector, auto-concurrency is prohibited as its status entails only one possible ongoing firing. With auto-concurrency and due to the potential difference in kernel execution times over different scenarios, tokens may "overtake" each other which makes it hard to assure determinacy [20]. Let us illustrate this using the example FSM-SADFG of Fig. 1. Inspired by [13], first we define the notion of a token sequence.

Definition 8 (Token sequence)**.** *Given a set* V *of values and a set* T *of tags used to model time, we view a token sequence* σ *as a member of the powerset* $2^{T \times V \times \mathcal{S}}$*, s.t.* $\pi_l(\sigma)$ *is order isomorphic to a subset of the integers, where* π_l *is the left projection function.*

E.g., $\langle (t, v, s) \rangle$ denotes a sequence containing one token produced at $t \in T$ time-units, with the value of $v \in V$ and produced by an actor operating in scenario $s \in \mathcal{S}$. We consider the execution of the FSM-SADFG of Fig. 1 from $t = 0$. In this case $V = \{s_1, s_2, *\}$ where $*$ denotes a token with arbitrary value, i.e. a data token, while $\mathcal{S} = \{s_1, s_2\}$. At $t = 0$, the only enabled process is the detector D. Assume that, by firing D, the FSM makes a transition from the initial state q_1 to state q_2. State q_1 corresponds to scenario s_1, and q_2 corresponds to s_2. After 1 time-unit, the control channels (D, A) and (D, B) host the token sequence $\langle (1, s_1, s_1) \rangle$. Channel (B, D) now hosts $\langle (0, *, \perp) \rangle$ which refers to the initial token i_3 as i_4 was just consumed by D firing. We use the $\perp$ notation to leave the scenario value unspecified as we do not know in which scenario initial tokens were produced. Now A can start firing by consuming the control token from channel (D, A), initial token i_8 from its self-edge and initial token i_6 from channel (B, A). The new status of (D, A) becomes $\langle\rangle$ and the new status of (B, A) becomes $\langle (0, *, \perp) \rangle$ that refers to initial token i_5. As there is still one initial token left on channel (B, D), D can perform its second firing. This results in channel (D, A) hosting $\langle (2, s_2, s_2) \rangle$, channel (D, B) hosting $\langle (1, s_1, s_1), (2, s_2, s_2) \rangle$ and channel (B, D) hosting $\langle\rangle$ with the assumption that the FSM transition (q_2, q_1) was taken. As A is auto-concurrent it can commence its second firing, but this time in scenario s_2 while still being busy with the first one in scenario s_1. Due to the difference in firing durations of A in scenarios s_1 and s_2, the firing of scenario s_2 started at $t = 2$ will finish earlier than the earlier firing of scenario s_1 started at $t = 1$. Therefore, at $t = 4$ the channel (A, B) will host the sequence $\langle (4, *, s_2), (4, *, s_2) \rangle$ as the firing duration

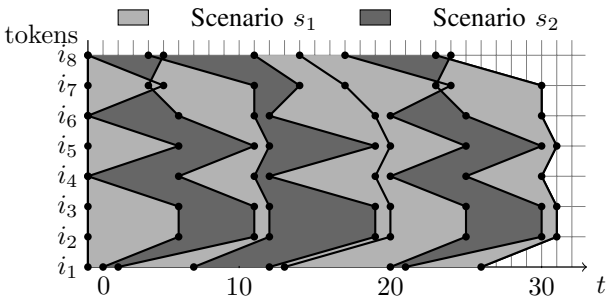


Fig. 2: Execution of the FSM-SADFG of Fig. 1

of A in scenario s_2 equals to 2 time-units. On the other hand channel (D, B) hosts the sequence $\langle (1, s_1, s_1), (2, s_2, s_2) \rangle$. Kernel B will therefore commence firing in scenario s_1 but by consuming the token $(4, *, s_2)$ produced by A in scenario s_2 and will not wait for the "right" scenario s_1 token that will be produced at $t = 5$. After B had completed this firing, the status of (B, B) becomes $\langle (5, *, s_1) \rangle$.

The "overtaking" phenomenon just discussed results in tokens being consumed in another scenario than the one they were produced in. This makes it hard to assure determinacy [20]. It is generally the modeller's responsibility to ensure that the model does not exhibit overtaking or that overtaking can be properly interpreted, i.e. that it does not invalidate the functional correctness of the system. It could however be handled with a policy in the semantics that ensures that tokens are only consumed by a kernel in the scenario they belong to.

In our work, we adopt the scenario-level FIFO policy of [9] inherent to the *(max,+)* semantics of FSM-SADF introduced in the same paper. One can view the execution of the FSM-SADFG of Fig. 1 as the execution of a sequence of SDF graphs obtained by applying the configurations of Fig. 1c and Fig, 1d to the scenario graph of Fig. 1a. In light of that observation, paper [9] considers scenarios in isolation as pure SDF graphs. By doing so, one avoids the overtaking problem as in a single scenario there can be no overtaking thanks to the static nature of SDF. At a later stage, scenarios are "glued" together and synchronized by the set of initial tokens. This can be done as initial tokens produced at the end of a scenario contain enough information to determine the timing of the next scenario [9]. The work of [9] assumes *self-timed execution*. Self-timed execution is a schedule where every actor fires as soon as possible, i.e. as soon as all required tokens are available. Fig. 2 shows the pipelined scenario execution of the FSM-SADFG of Fig. 1. Every scenario is initialized by the previous one. In Fig. 2, we see overtaking along the axes of availability times of tokens i_7 and i_8. E.g., we see that i_7 of scenario s_2 is actually produced (at $t = 4$) before i_7 of scenario s_1 (at $t = 5$), even though scenario s_1 was started first. In spite of this, the firing of B in scenario s_1 will be initialized by the availability of i_7 belonging to s_1 and not i_7 belonging to s_2 although it is produced first. Therefore, i_2 will be produced at $t = 6$ and not at $t = 5$ as discussed earlier. This way, determinacy is assured.

To introduce the *(max,+)* scenario-level FIFO policy to the semantics of Section III-A one has to find a way to decouple scenarios. This could be modeled by data channels having a

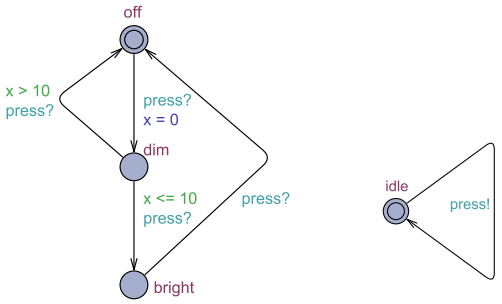
separate buffer for each scenario in the system and kernels only writing to and consuming from buffers belonging to the scenario they are currently operating in. For control channels such replication is not necessary as the detector is by definition "sequential", i.e. non auto-concurrent. With this concept of "scenario buffers", no overtaking between different scenarios is possible.

However, an interesting question emerges. In which scenario were the initial tokens produced? If we treat initial tokens as a special class (no scenario, buffer affiliation) we will easily violate the functional correctness of the system, e.g. introduce a deadlock. In the scenario graph of Fig. 1 imagine that the detector has fired twice by consuming initial tokens i_3 and i_4 following the path $q_1 \rightarrow q_1 \rightarrow q_1$ of the FSM. This means that the graph has executed the scenario sequence $s_1 s_1$. To complete the sequence, actor B has fired twice so channel (B, D) hosts the sequence $\langle (t_1, *, s_1), (t_2, *, s_1) \rangle$. Now, if the transition $q_1 \rightarrow q_2$ of the FSM is to be taken, the resulting (B, D) channel state after the completion of the scenario will be $\langle (t_2, *, s_1), (t_3, *, s_1) \rangle$. Being in state q_2, the FSM can only perform the transition $q_2 \rightarrow q_1$. By the FIFO policy, to do that tokens belonging to s_2 need to be consumed. However, channel (B, D) only hosts tokens belonging to s_1. Therefore, a deadlock occurs.

Another approach might be to force initial token scenario affiliation. However, this approach would also violate the functional correctness of the model by introducing a deadlock or by restricting the language the scenario FSM accepts.

As it is not clear how to deal with initial tokens belonging to channels on which overtaking can take place, we only allow overtaking on channels with no initial tokens, i.e. auto-concurrent actors can only produce in data buffers that are initially empty. This excludes actor self-edges as these are used to limit actor's auto-concurrency by assigning them an appropriate number of initial tokens. Actually, in the UPPAAL model of FSM-SADF of Section IV they will be modelled as the number of instances of a particular actor in the system. Consequentially, we only replicate data buffers that are filled by auto-concurrent actors and are initially empty. Under this restriction on the structure of the input FSM-SADF specification, the FIFO policy can be straightforwardly encoded into the semantics of Section III-A by changing the definition of the set of buffers of Definition 1 to $\mathcal{B} \subseteq \mathcal{K} \times \mathcal{P} \times (\mathcal{S} \cup \mathbf{s})$ where $\mathbf{s}$ is the "default scenario", which marks the buffer used when there is no overtaking on the channel, i.e. the default buffer. Also, we redefine $In(p, s) = \{b \in \mathcal{B} \mid \pi_2(b) = p \wedge (\pi_3(b) = s \text{ when } \omega(b) = 1 \,;\, \mathbf{s} \text{ otherwise})\}$, where π_2 and π_3 are the 2nd and 3rd projection function, respectively and $\omega : \mathcal{B} \rightarrow \{0, 1\}$ is the function returning the information whether overtaking is possible on the channel implemented by buffer b. Similarly, $Out(k, s) = \{b \in \mathcal{B} \mid \pi_1(b) = p \wedge (\pi_3(b) = s \text{ when } \omega(b) = 1 \,;\, \mathbf{s} \text{ otherwise})\}$, where π_1 is the 1st projection function. Last, kernel start and end actions must be altered to fit the new definitions. E.g. the kernel start action transition is enabled if $|\psi(k)| \geq 1$ and $\forall b \in In(k, s) : \phi(b) \geq R_c(b, s)$.

Another, simpler option is to disallow auto-concurrency as done in [11], [20]–[22] for SADF. In this case, the notion of a multiset of ongoing kernel firings in the kernel status of Definition 2 has to be changed so each kernel can have zero or one ongoing firings: $\kappa : \mathcal{K} \rightarrow (\mathcal{S} \times \mathbb{N}_0) \cup \{-\}$. To reflect this re-definition, the kernel start and end actions have to be re-adjusted in the spirit of detector start and end actions (recall that the detector is "sequential" by definition).



(a) Light (b) User

Fig. 3: Network of timed automata in UPPAAL

IV. Translation of FSM-SADF to Timed Automata

To be able to model check an FSM-SADF specification, we encode the the operational semantics of FSM-SADF in the UPPAAL model checker. The correctness of the translation follows from the construction itself as explained in the remainder of this section. We limit our attention to self-timed bounded FSM-SADF specifications [17]. [1]

In UPPAAL, a system is modeled as a network of TA that is extended with bounded discrete variables that are part of the state. These variables can be read, written and are subject to common arithmetic operations.

We recall the definition of TA where we use $\mathcal{B}(\mathcal{C})$ to denote the set of constraints defined over a finite set of real-valued variables $\mathcal{C}$ called *clocks* and where $\Sigma = \{a!, a?, \ldots\}$ is a finite alphabet of synchronization actions.

Definition 9 (Timed automaton (TA))**.** *A timed automaton $\mathcal{A}$ is a tuple (L, l^0, E, I), where L is a finite set of locations (nodes), l^0 is the initial location, $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times L$ is the set of edges between locations with a guard, an action and a set of clocks to be reset, and $I : L \rightarrow \mathcal{B}(\mathcal{C})$ assigns invariants to locations. We shall write $l \xrightarrow{g,a,r} l'$ when $(l, g, a, r, l') \in E$.*

A state of the system modeled in UPPAAL is defined by the locations of all automata, the clock values, and the values of the discrete variables. Every automaton may fire an edge (sometimes misleadingly called a transition) separately or synchronise with another automaton, which leads to a new state [4]. An example of a network of timed automata is shown in Fig. 3. The network models a time-dependant light-switch (Fig. 3a) and its user (Fig. 3b). The switch and the user communicate using the `press` labels (channel). The user can press the switch (`press!`) randomly at any time or even not press the switch at all. The switch waits to be pressed (`press?`). If the user presses the switch, the light is on, but dimmed (location **dim**). If the user presses the switch again, but after more than 10 time-units (guard `x > 10`), the light is off (location **off**). If the user presses the switch within 10 time-units (guard `x <= 10`) the light is brightened (location

[1] The UPPAAL models of all FSM-SADF graphs in this paper, as well as the SDF^3 MPEG-4 decoder, can be found at https://bitbucket.org/tasadf/models

bright). At this point, whenever the user presses the switch, the light will turn off (location **off**).

The FSM-SADF configuration of Definition 2 is modeled so that the kernel and detector statuses are encoded in the states of the TA, while the buffer and control statuses are modelled explicitly using discrete variables. These discrete variables are read and written during kernel/detector start/end actions. Operations performed on discrete variables correspond to checking the availability of input tokens, token consumption and token production. Discrete variables do not add to the expressive power of the formalism, and for presentation purposes, we do not encode their use in the TA edge firings.

Given an FSM-SADFG G, we generate a parallel composition of TA $System = \mathcal{A}_{k_1}^{\|\gamma(k_1)} \parallel \ldots \parallel \mathcal{A}_{k_n}^{\|\gamma(k_n)} \parallel \mathcal{A}_d$, where $k_i \in \mathcal{K}$ and $n = |\mathcal{K}|$. By $\mathcal{A}_{k_i}^{\|\gamma(k_i)}$ we denote the fact that $\gamma(k_i)$ TA in parallel are used to model kernel k_i. Function $\gamma : \mathcal{K} \to \mathbb{N}$ gives the realized auto-concurrency of a kernel. If the kernel k_i has a self-edge, i.e. $(k_i, k_i) \in \mathcal{B}$, then $\gamma(k_i) = \phi_\iota((k_i, k_i))$. If kernel k_i has no self-edge, $\gamma(k_i)$ can be found experimentally. We assume some N_{k_i}, then we need to verify that the actual $\gamma(k_i)$ is strictly smaller, i.e., $\gamma(k_i) < N_{k_i}$. This is discussed in Section V.

Fig. 4 shows the UPPAAL model of the FSM-SADFG of Fig. 1. In the description language of UPPAAL, processes are obtained as instances of parametrized process templates. In our translation we define two templates: The kernel template of Fig. 4a and the detector template of Fig. 4b. The kernel template is generic, while the detector template is customized to correspond to the FSM of the input FSM-SADF specification. Note that control buffers are implemented as FIFOs where the values of FIFO elements are the scenario IDs, while data buffers are abstracted into integers as only the amount of data buffer tokens matters, not their value.

Every kernel $k_i \in \mathcal{K}$ is translated to the TA $\mathcal{A}_{k_i} = (L_i, l_i^0, E_i, I_i)$ where $L_i = \{\texttt{Initial}, \texttt{Fire}\}$, $l_i^0 = \texttt{Initial}$, and E_i and I_i are given as follows. The edge

$$\texttt{Initial} \xrightarrow{|\psi(k_i)|\geq 1 \wedge \forall b \in In(k_i): \phi(b) \geq R_c(b,s), \emptyset, \{x_i\}} \texttt{Fire}$$

corresponds to the kernel start action. To start firing, the kernel must first gain knowledge in which scenario is it operating in. This information is stored in the kernel's control buffer. Therefore, the kernel *peeks* into its control buffer if it is not empty, finds out the operating scenario s and waits for the availability of the required number of tokens in its data buffers. This behaviour is encoded using the guard `bool k_tok_available(int ker_id)` where `ker_id` is the ID of the kernel. Once the guard evaluates to `true`, the kernel can actually perform the start action, by consuming input tokens both from its control buffer and its data buffers. Consumption corresponds to statuses being decremented. This behaviour is encoded by the function `void k_start_fir(int ker_id, int& scen_id, int& delay)`, where `scen_id` is the ID of the operating scenario, and `delay` is the kernel's firing duration in the operating scenario. Aforementioned variables get their values within the update label although these are known during the evaluation of the `k_tok_avail` guard. This is because guards in UPPAAL must be side-effect free. These variables are needed by the kernel end action that corresponds to the edge

$$\texttt{Fire} \xrightarrow{x_i = E(k_i,s), \emptyset, \emptyset} \texttt{Initial}$$

and the invariant $I(\texttt{Fire}) = x_i \leq E(k_i, s)$. These two assure that the system stays in the location `Fire` for exactly the execution time $E(k_i, s)$ of the kernel k_i in scenario s. Thus, time transitions are encoded implicitly in the operation of the network of TA, for which time progresses in unison. The resulting data token production is coded in the function `void k_end_fir(int ker_id, int scen_id)`.

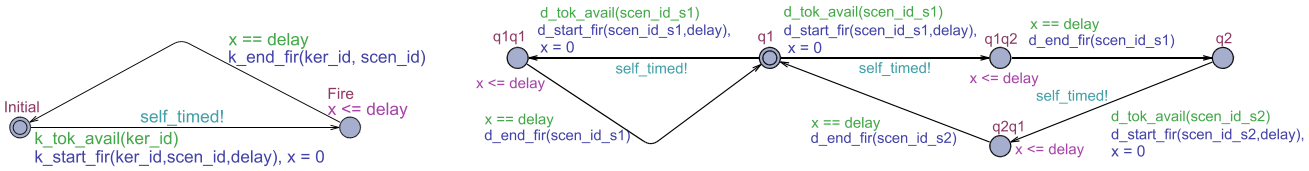
The detector TA uses the structure of its FSM (e.g. locations `q1` and `q2` of Fig. 4b correspond to states q_1 and q_2 of the FSM of Fig. 1b), but embeds in each transition a firing location wherein time can pass between the events of consuming the input tokens and producing the output tokens. We encode it as $\mathcal{A}_d = (L_d, l_d^0, E_d, I_d)$, where $L_d = \mathbb{S} \cup \{(q, q') \mid q, q' \in \mathbb{S} \wedge q' \in \mathbb{T}(q)\}$ and $l_d^0 = \iota$. The edge set E_d is defined such that each transition $q_i \to q_j$ described by $\mathbb{T}$ is translated into a detector start edge followed by a detector end edge:

$$\mathtt{q}_i \xrightarrow{\forall b \in In(d): \phi(b) \geq R_c(b, \Phi(q_i)), \emptyset, \{x_d\}} (\mathtt{q}_i, \mathtt{q}_j) \xrightarrow{x_d = E(d, \Phi(q_i)), \emptyset, \emptyset} \mathtt{q}_j$$

The invariant function I_d is defined such that each firing location $(\mathtt{q}_i, \mathtt{q}_j)$ maps to the invariant $x_d \leq E(d, \Phi(q_i))$. The guard `bool d_tok_avail(int scen_id)` of edge $\mathtt{q}_i \to (\mathtt{q}_i, \mathtt{q}_j)$ assures that there are enough tokens present in detector's input data buffers before it commences firing. The operating scenario of the detector depends on the current state of the scenario FSM. The function `void d_start_fir(int scen_id, int& delay)` updates data buffer statuses as the result of the detector start action being performed. The `delay` variable receives its value within the function. The update function `void d_end_firing(int scen_id)` of edge $(\mathtt{q}_i, \mathtt{q}_j) \to \mathtt{q}_j$ encodes the effects of the detector end action to control statuses, i.e. the production of control tokens.

To assure determinacy in the presence of auto-concurrency we revert to the considerations of Section III-B. In UPPAAL, this means that we replicate data buffers which are being filled by auto-concurrent kernels over scenarios by simply declaring them as arrays of integers, where the array index corresponds to the scenario ID. The aforementioned guard and update functions of Fig. 4 all use `scen_id` as the input parameter and will therefore operate on the correct buffer replicas. As mentioned in Section III-B, auto-concurrency is only allowed for kernels that produce to buffers with no initial tokens. Buffers in which overtaking is not possible, are not replicated and the "default container" is used to store the number of tokens present in the buffer. Whether overtaking is possible or not in a buffer is encoded with a configuration constant that is checked in guard and update functions.

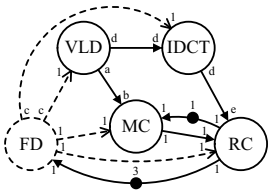
When it comes to scheduling policies, the operational semantics of Section III-A does not prescribe a particular one and consequentially neither does the previously discussed TA translation. However, it is often convenient to assume a certain class of scheduling policies. An important class are those policies where actions take place without delay, i.e. processes fire as soon as they are enabled. We refer to executions under such policies as *self-timed executions*. In UPPAAL, the concept of urgency can be exploited to impose such a policy.



(a) Kernel in UPPAAL

(b) Detector in UPPAAL

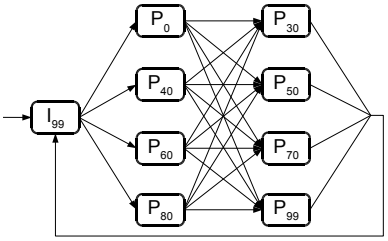
Fig. 4: UPPAAL model of the FSM-SADFG of Fig. 1



Rate	Scenario I_{99}	P_0	P_x
a	0	0	1
b	0	0	x
c	99	1	x
d	1	0	1
e	99	0	x

$x \in \{30, 40, 50, 60, 70, 80, 99\}$

(a) Scenario graph



(b) Scenario FSM

Fig. 5: FSM-SADF model of an MPEG-4 decoder [17], [18]

Specifically, an urgent broadcast channel [4] can be used to force kernel and detector start actions without delay (channel `self_timed` in Fig. 4). Following the work of [9], [11], [21], [22] we adopt the concept of self-timed execution in the remainder of this paper.

To conclude, the translation of this section enables the user to represent an FSM-SADF specification as a network of TA. The user can chose between allowing or prohibiting auto-concurrency, by the use of system-level declarations. In case auto-concurrency is enabled, the user can chose to use or not to use the scenario-level FIFO policy of [9] to assure determinacy. Furthermore, the user can choose to consider self-timed execution or not by the use of the urgency concept in UPPAAL.

V. Model Checking of TA Model

In this section we demonstrate examples of qualitative and quantitative analysis of FSM-SADF specifications using the UPPAAL model checker and its query language. UPPAAL's query language is used to specify the properties to be checked and is a subset of TCTL (timed computation tree logic) [2]. We use the MPEG-4 decoder of Fig. 5 as our case-study [17], [18]. All parameters in our case study are taken from [18].

The decoder functionality is given by the scenario graph of Fig. 5a. The decoder processes streams consisting of I and P frames. These frames consist of a variable number of macro-blocks (0 to 99 for QCIF). The detector (FD) detects the frame type. If the frame type is I, all frame macro-blocks are decoded by the VLD and IDCT kernels, while the image is reconstructed by the RC kernel. When the frame type is P, motion compensation is required in the decoding process. Its functionality is implemented by the MC kernel. I frame processing defines the I_{99} scenario, while P frame processing defines the P_x scenarios where x stands for the number of macro-blocks in the P frame and $x \in \{0, 30, 40, 50, 60, 70, 80, 99\}$. The scenario occurrence pattern is defined by the scenario FSM of Fig. 5b.

FSM-SADF is a non-deterministic model. With non-deterministic models we are interested in worst-case analysis. The particular metrics analyzed, the obtained results and the associated time and memory usage are shown in Table I. We compare our results and resource requirements to those of the SDF^3 tool. In SDF^3, two sets of algorithms can be used to analyze FSM-SADF: 1) The algorithms of the customized model checker [19], [21] of SADF by considering states for which the involved reward is maximal. However, these algorithms can be used only if there is no auto-concurrency in the FSM-SADFG (e.g. all kernel have self-edges with one initial token). In that case, the graph can be analyzed for deadlock freedom, maximum buffer occupancy, maximum inter-firing latency, maximum response delay and throughput. 2) The *(max,+)* based algorithms of [9] specifically designed for FSM-SADF that can only analyze throughput and latency. The *(max,+)* techniques for what they offer (throughput and latency) will outperform their model-checking counterparts. However, the model-checking based SADF techniques offer a wider palette of metrics amenable to analysis. Therefore, we base our comparison on the techniques of SADF. To be able to compare our results to those of SDF^3, we limit the auto-concurrency of all kernels in Fig. 5a to one (imagine that all kernels have a self-edge with one initial token). Before proceeding, we point out a subtle difference between the operational semantics of SADF and FSM-SADF. In contrast to SADF model, in FSM-SADF the value of the control tokens produced by the detector end action depends on the current state of the FSM and not on the next state. Therefore, to obtain the same behaviour and corresponding analysis results, the FSM of Fig. 5b needs to be augmented with the one additional state I' before being subjected to SDF^3 analysis. State I' must be declared initial with one outgoing transition leading to the "original" initial state I.

The experiments were performed on an Intel Core i5-750 CPU with 8 GB of main memory running UPPAAL 4.1.19 64-bit on Linux. The default settings were used and UPPAAL was restarted between each query.

TABLE I: MPEG-4 verification time and virtual memory usage

Analysis	SDF^3 Result	SDF^3 Time [s]	SDF^3 Mem [MB]	UPPAAL Result	UPPAAL Time [s]	UPPAAL Mem [MB]
Deadlock freedom	Deadlock free	n/a	n/a	Deadlock free	17.07	564
Maximum buffer occupancy, all buffers	278, 278, 3, 3, 2, 99, 1, 1, 116, 3	17.27	169	278, 278, 3, 3, 1, 99, 1, 1, 108, 3	7.31	551
Maximum inter-firing latency, all processes	4327, 7470, 40, 4303, 40	6.44	171	4327, 7470, 40, 4327, 57	11.45	557
Maximum response delay, all processes	0, 0, 40, 4327, 57	0.08	85	0, 0, 40, 4327, 57	0.94	312
Throughput	0.000363636	1.10	86	n/a	n/a	n/a
Process interleavings, MC & RC	n/a			1	8.34	556
Inter-process delay, MC & RC	n/a			≤ 5000	13.61	561
Realized kernel auto-concurrency, VLD	n/a			2	31.84	1072
Pipeline depth	n/a			3	7.30	551
Parallelism between actors, MC & RC	n/a			None	7.29	551

a) Deadlock: Analyzing for deadlock freedom is achieved with the UPPAAL query (`A[] not deadlock`). In SDF^3, checking whether a FSM-SADFG is deadlock free is preformed during the computation of any metric, i.e. there is no special option for this type of analysis and the respective time and memory requirements cannot be given (n/a in Table I).

b) Maximum buffer occupancy: Maximum buffer occupancy of a particular buffer under all possible self-timed schedules is obtained using the UPPAAL supremum operator, e.g. (`sup:` b_i). In SDF^3, we use the input argument `--compute buffer_occupancy"(maximum)"` during application invocation. The results are ordered by the sequence: FD2VLD, FD2IDCT, FD2MC, FD2RC, VLD2IDCT, VLD2MC, RC2MC, MC2RC, IDCT2RC, RC2FD. The difference in results between SDF^3 and UPPAAL is due to another subtle difference in the operational semantics of SADF and FSM-SADF. In FSM-SADF, token consumption takes place during detector/kernel start actions, while in SADF consumptions take place at a later point - during the detector/kernel end actions. Therefore, SDF^3 will often deliver higher values.

c) Maximum inter-firing latencies: Maximum inter-firing latency of a process is defined as the maximum elapsed time between two successive firing completions of the process. In UPPAAL, process latencies can be obtained as suprema of the clock `y` that is reset every time process p completes its firing, i.e. `sup:` p`.y` while p`.y = 0` every time edge p.`Fire` $\rightarrow$ p.`Initial` of Fig. 4a is fired for the kernels or every time edge $d.(\mathrm{q}_i, \mathrm{q}_j) \rightarrow d.\mathrm{q}_j$ of Fig. 4b is fired for the detector. In SDF^3, we use the input argument `--compute inter_firing_latency"(maximum)"`. The results are sequenced by FD, MC, VLD, RC, IDCT. In our experiments we assume that all processes have just competed firing at $t = 0$, while SDF^3 makes no such assumption. Therefore, the difference in values delivered by SDF^3 and UPPAAL is merely of "syntactical nature".

d) Maximum Response Delays: Maximum response delay of a process denotes the maximum time until the first firing completion of that process. In UPPAAL, we determine it by checking the relationship between the maximum response delay of a process p and a constraint r using the query (`E<>` p`.bFirstFirCompleted and` p`.y >=` r), where p`.y` is a clock that is never reset, and p.`bFirstFirCompleted` is a variable set to `true` when p completes its first firing. In SDF^3, we use the input argument `--compute response_delay"(maximum)"`.

e) Throughput: Throughput of a process is defined as the long-run average number of firing completions of a process per-time unit. In SDF, the throughput of the entire graph is defined as the throughput of a process normalized (divided) by the number of firings of that process within the graph iteration [10]. The same definition can be applied to FSM-SADF. Furthermore, as in FSM-SADF the repetition vector entry of the detector equals to one for all scenarios (detector fires once per scenario), so is the throughput of the entire graph equal to the throughput of the detector process. The TCTL [2] based query language of UPPAAL cannot be used to evaluate such long-run averages. In SDF^3, we use the input argument `--compute throughput"(FD)"`.

f) Process interleaving: We continue with a set of simple reachability properties not supported by SDF^3, but that can easily be verified in UPPAAL.

We check the interleaving of different process firings, e.g. "between two consecutive firing completions of the process p, process q completes at least n firings", etc. For this we use a *leads to* query (whenever a eventually b) and a counter variable: (p`.Fire --> `q`.FireCount >=` n). Variable q`.FireCount` is reset every time p takes the edge `Fire` $\rightarrow$ `Initial` and incremented every time q takes the same edge. In the experiment, $p =$ MC, $q =$ RC, and $n = 1$.

g) Inter-process delay: We can also check whether the maximum delay between the completion times of firings of two processes within a scenario is greater than, less than or equal to a predefined value by constructing a query monitor TA that synchronizes with the events of firing completions of the processes it monitors. In the case of a kernel p, this synchronization takes place when the edge p.`Fire` $\rightarrow$ p.`Initial` is taken. In the experiment we verify that the MC-RC delay is always smaller than 5000.

h) Realized kernel auto-concurrency: If we assume auto-concurrency up to the level N_p for a particular kernel p, we can check whether it has been fully utilized or not. Allowing N_p concurrent executions of p means that we have assigned N_p processing elements to p. If all are not used, the idle ones can be assigned to kernels of another application. This design decision might improve the overall performance of the system hosting multiple applications. To determine the realized auto-concurrency of a kernel p we use the query `sup:` $p.count$

where $p.count$ is a variable that is incremented every time edge `Initial` $\rightarrow$ `Fire` is taken and decremented whenever the edge `Fire` $\rightarrow$ `Initial` is taken. In this experiment, $p = VLD$, $N_{VLD} = 2$ and `sup` : $VLD.count = 2$.

i) Pipeline depth: Pipeline depth denotes the maximum number of scenario executions active at the same time. In case of our MPEG-4 case study, the beginning of a scenario is marked by the firing completion of the detector FD, while its end is marked by the firing completion of the RC process. To compute the pipeline depth $pdepth$, use the query `sup` : $pdepth$ where $pdepth$ is incremented every time FD takes the edge $(\mathtt{q}_i, \mathtt{q}_j) \rightarrow \mathtt{q}_j$ and decremented every time RC takes the `Fire` $\rightarrow$ `Initial` edge. In this case the pipeline depth is three as there are three initial tokens in the data buffer (RC, FD). This was immediately visible in this example, but one can easily imagine a more complicated initial token distribution where mere visual inspection would not suffice.

j) Parallelism between actors: We can check whether processes in an arbitrary subset of $\mathcal{P}$ can fire in parallel. E.g., processes p and q can fire in parallel if the query `E<>` p`.Fire and` q`.Fire` evaluates to `true`. In our experiment $p = MC$ and $q = RC$. These two cannot fire in parallel.

VI. Discussion

Another look at Table I reveals that UPPAAL allows us to check the model against various properties, many of which are not supported by the SDF^3 tool-set. On average, UPPAAL analysis will take the same time as that of SDF^3, but with higher memory demands. This observation justifies the use of a general verification tool such as UPPAAL as a complement to specialized tools. The flexibility of the UPPAAL's TCTL based query language and the possibility of construction of various query monitor automata allows the user to easily compute various qualitative and quantitative properties of the model.

The only metric not supported by UPPAAL that is available in SDF^3 is throughput. Here we take the opportunity to shortly discuss the applicability of UPPAAL and TCTL in a conservative estimation of the throughput value.

Let $\mathcal{W}$ be a window of time of finite duration W. The throughput equals to the long run average number of firing completions of the detector process per time-unit. Therefore, let c be a variable that is incremented every time the detector completes a firing, i.e. takes the $(\mathtt{q}_i, \mathtt{q}_j) \rightarrow \mathtt{q}_j$ edge and reset every time when clock T equals to W ($T == W$) along with the clock T itself. By defining the property `A[]` ($T == W$) `imply` $c >= H$, we can verify that the value of c within **all** time windows $\mathcal{W}$ will be greater or equal than the value of the *horizon* H. If the property is satisfied, the conservative (property holds along the entire time axis divided into windows $\mathcal{W}$ of duration W) throughput estimate $\tilde{\rho}$ is given by $\tilde{\rho} = H/W$. By using larger values of W and by finding the maximum H for which the property holds, we tighten the estimate at the price of increased analysis time. We call this method of conservative throughput estimation for FSM-SADF the *horizon method*. Unfortunately, the horizon method poorly scales, and we could not apply it successfully to our MPEG-4 case study. However, it was applicable to the "small" example FSM-SADF graph of Fig. 1. Fig. 6 shows how $\tilde{\rho}$ converges to the exact value of 0.25 for growing W. Some points are decorated with the time required by UPPAAL to deliver the estimate value.

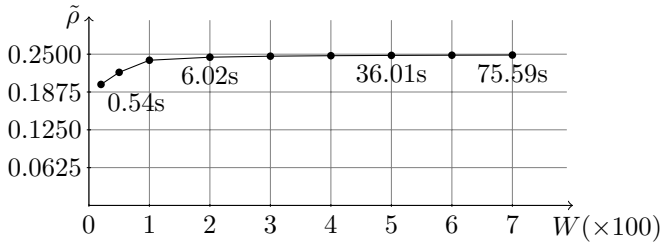


Fig. 6: Convergence of the horizon method for throughput estimation for the FSM-SADFG of Fig. 1

TABLE II: UPPAAL vs. SDF^3 scalability via maximum buffer occupancy

	SDF^3		UPPAAL	
Pipeline depth	**Time [s]**	**Mem [MB]**	**Time [s]**	**Mem [MB]**
1	0.80	90	1.14	121
2	5.5	113	2.96	196
3	17.96	169	7.42	567
4	64.66	319	20.44	1144
5	392.02	662	81.17	2551
6	$>$ 1800	$>$ 1708	$>$ 133	$>$ 5752

TABLE III: UPPAAL scalability via maximum buffer occupancy for increased auto-concurrency

	UPPAAL	
Number of VLD instances	**Time [s]**	**Mem [MB]**
1	7.48	559
2	31.44	1088
3	$>$ 175.00	$>$ 6249

VII. Scalability Issues

We investigate the scalability of the TA FSM-SADF analysis from two angles.

First, we consider time and memory requirements for the maximum buffer occupancy calculation for the MPEG-4 case study while increasing the pipeline depth, i.e. the number of initial tokens on (RC, FD) channel. The results and the comparison with the SDF^3 tool are shown in Table II. With the pipeline depth of six, both tools experience state-space induced complexity problems. Among the metrics supported by SDF^3, the maximum buffer occupancy is the most demanding computation as the diamond property of the underlying TPS [21] cannot be exploited, i.e. all interleavings of timeless actions need to be considered. UPPAAL will by default considers all possible interleavings of timeless actions and will time-wise perform better that SDF^3 for this type of analysis.

Second, for UPPAAL, we check how it copes with kernel-level auto-concurrency when performing the maximum buffer occupancy analysis. We perform the experiments by increasing the number of VLD kernel instances in the model. The results of the experiments are shown in Table III. Already with three instances of VLD we experience state-space induced complexity problems. This indicates to relatively poor scalability

of the translation in the presence of auto-concurrency when performing maximum buffer occupancy analysis.

It is not clear how to improve the scalability of the model for maximum buffer occupancy analysis and other analysis defined by queries over discrete variables. This is because in these type of analysis all action interleaving need to be considered. However, for the analysis of temporal properties like inter-firing latency and process response times (model-checking against clocks), the SADF diamond property could be exploited. Unfortunately, traditional partial order reduction techniques embedded in UPPAAL are insufficient to take advantage of this. However, process priorities of UPPAAL could be used to do exactly that by arbitrarily prioritizing process timeless actions. This way significant state space reductions could be achieved. We leave these considerations to future work.

VIII. Conclusion and Future Work

FSM-SADF is a powerful dataflow formalism that is able to capture the dynamic behaviour of modern streaming applications while offering a good trade-off between expressiveness, analyzability and implementation efficiency. However, the formalism is currently only supported by the SDF^3 toolset which implements a predefined set of properties that can be analysed/verified. In this paper we propose a translation of FSM-SADF to TA, thereby enabling the use of the UPPAAL model checker for analysing and verifying user-defined properties in a straightforward manner. Our translation of FSM-SADF model to TA is also the first translation of a member of the SADF MoC family to a model-checker that supports auto-concurrency. We also report on the scalability issues experienced. As future work we plan to improve the scalability of the translation in the analysis of temporal properties of the model by using priorities in UPPAAL. Furthermore, as our translation also sets the first milestone towards enabling the use of FSM-SADF in a wider context, e.g. cost-optimal analysis, we also plan to investigate reachability analysis of applications modeled by FSM-SADF through an optimal control formulation using the UPPAAL family of model-checkers.

Acknowledgement

This work is supported by the 7th EU Framework Program under grant agreement 318490 (SENSATION).

References

[1] W. Ahmad, R. de Groote, P. Holzenspies, M. Stoelinga, and J. van de Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In *Application of Concurrency to System Design (ACSD), 2014 14th International Conference on*, pages 72–81, June 2014.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425, Jun 1990.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, Apr. 1994.

[4] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*. Springer, 2004.

[5] S. S. Bhattacharyya, E. F. Deprettere, and B. D. Theelen. Dynamic dataflow graphs. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 905–944. Springer, 2nd edition, 2013.

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

[7] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Throughput-constrained DVFS for scenario-aware dataflow graphs. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 175–184, April 2013.

[8] M. Fakih, K. Gruttner, M. Franzle, and A. Rettberg. Towards performance analysis of SDFGs mapped to shared-bus architectures using model-checking. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1167–1172, March 2013.

[9] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, Oct 2010.

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

[11] J.-P. Katoen and H. Wu. Exponentially timed SADF: Compositional semantics, reductions, and analysis. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10, Oct 2014.

[12] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), Sept 1987.

[13] E. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, Dec 1998.

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

[15] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *System on Chip (SoC), 2011 International Symposium on*, pages 14–21, Oct 2011.

[16] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD 2006), 28-30 June 2006, Turku, Finland*, 2006.

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

[18] S. Stuijk, A. Ghamarian, B. Theelen, M. Geilen, and T. Basten. FSM-based SADF. Technical report, Eindhoven University of Technology, Department of Electrical Engineering, 2008.

[19] B. Theelen. A performance analysis tool for scenario-aware streaming applications. In *Quantitative Evaluation of Systems, 2007. QEST 2007. Fourth International Conference on the*, pages 269–270, Sept 2007.

[20] B. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, July 2006.

[21] B. D. Theelen, M. Geilen, and J. Voeten. Performance Model Checking Scenario-Aware Dataflow. In *Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 6919 of *Lecture Notes in Computer Science*, Aalborg, Denmark, 2011. Springer.

[22] B. D. Theelen, J.-P. Katoen, and H. Wu. Model checking of Scenario-Aware Dataflow with CADP. In W. Rosenstiel and L. Thiele, editors, *Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, 2012. IEEE.