Resource-Constrained Optimal Scheduling of Synchronous Dataflow Graphs via Timed Automata

Waheed Ahmad, Robert de Groote, Philip K.F. Hölzenspies, Mariëlle Stoelinga, Jaco van de Pol
University of Twente, The Netherlands
Email: {w.ahmad, e.deGroote, p.k.f.holzenspies, m.i.a.stoelinga, j.c.vandepol}@utwente.nl

***Abstract*—Synchronous dataflow (SDF) graphs are a widely used formalism for modelling, analysing and realising streaming applications, both on a single processor and in a multiprocessing context. Efficient schedules are essential to obtain maximal throughput under the constraint of available resources. This paper presents an approach to schedule SDF graphs using a proven formalism of timed automata (TA). TA maintain a good balance between expressiveness and tractability, and are supported by powerful verification tools, e.g. UPPAAL. We describe a compositional translation of SDF graphs to TA, and perform analysis and verification in the UPPAAL state-of-the-art tool. This approach does not require the (exponential) transformation of SDF graphs to homogeneous SDF graphs and helps to find schedules with a trade-off between the number of processors required and the throughput. It also allows quantitative model checking and verification of (preservation of) user-defined properties such as the absence of deadlocks, safety, liveness and throughput analysis. This translation also forms the basis for future work to extend this analysis of SDF graphs with new features such as stochastics, energy consumption and costs.**

I. INTRODUCTION

Modern multimedia applications, such as multi-party video conferencing and video-in-video, impose high demands on a system's throughput. At the same time, resource requirements (buffer sizes, number of processors used) should be minimised. Therefore, smart scheduling strategies are needed.

Synchronous Dataflow (SDF) graphs are well-known computational models for analysing dataflow and digital signal processing applications and are increasingly utilised for both modelling and analysing multimedia applications on a multiprocessor Systems-on-Chip (MPSoC) [16]. In this paper, all software tasks of an application are modelled as SDF actors.

Currently, resource-allocation strategies and scheduling of tasks for SDF graphs are carried out using the max-plus algebraic semantics and graph analysis by transforming SDF graphs to equivalent Homogeneous SDF graphs (HSDF) [5] [12]. This approach leads to a larger graph; in the worst case, the derived HSDF graph can be exponentially larger than the original SDF graph [20]. Our approach does not avoid this exponential complexity but rather we are solving the problem *with* resource constraints in the *same* complexity.

Another state-of-the-art method [9] calculates the throughput of SDF graphs by exploring the state-space until a periodic phase is found. However, in this method, each actor is executed as soon as it is enabled and it is assumed that sufficient resources are available to accommodate all the enabled executions simultaneously. On the contrary, this may not be the case in real-life applications, where there is always a constraint on the number of resources.

We propose an alternative, novel approach to analyse schedules of SDF graphs on a limited number of processors using Timed Automata (TA) [3]. TA are a natural choice for modelling time-critical systems to check whether the timing constraints are met. By definition, TA are automata in which clock variables measure the elapse of time. Clock guards on the edges indicate conditions under which an edge can be taken and invariants show the conditions under which a system can stay in a certain location. TA are extensively used in the verification and model checking of industrial applications [17].

In particular, our main contributions are: (1) Translating SDF graphs into timed automata in a compositional manner; (2) Exploiting UPPAAL's [4] capabilities to search the state-space and derive a schedule that fits on the given number of processors while maximising throughput; (3) Handling heterogeneous processor models, in which only specific processors can run a particular task. In this way, we can efficiently determine a trade-off between the number of processors and the throughput for a certain application. This will hugely aid in finding efficient schedules in terms of energy and memory consumption. We also demonstrate that our translation preserves deadlock freedom if the number of processors varies.

Quantitative model checking and support for evaluating user-defined properties is lacking in the existing contemporary SDF graph analysis tools e.g. SDF^3 [22]. In this context, UPPAAL is exploited to address this lack and to evaluate user-defined properties which further adds to the benefits of SDF graphs. Future research directions are to carry on from the results achieved in this paper and explore the possibilities of extending the analysis of SDF graphs with the new features, i.e. stochastics and energy costs and combine with new extensions of TA like costs and timed games.

Paper organisation. Firstly, Section II reviews related work. Section III explains SDF graphs and Section IV discusses the throughput analysis of SDF graphs and our method of calculating it. Section V covers TA and UPPAAL and Section VI covers the translation of SDF graphs to TA. The methodology of analysing SDF graphs using UPPAAL is explained in section VII. Section VIII experimentally validates our approach via case studies. Finally, Section IX draws conclusions and outlines possible future research.

II. RELATED WORK

Various dataflow models exist, such as computational graphs [12] and SDF graphs [16]. SDF graphs are the more expressive of the two, and can analyse applications running on multiprocessors, such as MPEG-4 and MP3 decoder. Minimising the buffer requirements of SDF graphs using model checking is analysed in depth [8], [10]. Throughput analysis of HSDF graphs is studied extensively in [5], [12], [25], [27]. An algorithm proposed by Karp in [12] to calculate maximum cycle ratio (MCR) is another efficient method of calculating the throughput. All these studies

1550-4808/14 $31.00 © 2014 IEEE
DOI 10.1109/ACSD.2014.13

require a conversion of SDF graphs into HSDF graphs [16], [27] which can be exponentially larger than the original SDF graphs in the worst case. On the other side, the throughput calculation method applicable directly to SDF graphs [9] is practical only if we have sufficient processors. However, our strategy calculates maximal throughput on a given finite number of processors.

Another novel technique for task binding and scheduling of SDF graphs under given throughput constraints is presented in [20]. But this approach uses an combination of static-order and TDMA scheduling for actors within an application, unlike in our strategy where the actors are mapped in such a way that maximal throughput is achieved at run-time. A model-checking based approach to guarantee timing bounds of multiple SDF graphs running on a shared-bus multicore architectures is analysed in [6]. However, this analysis also requires a static-order scheduling.

Model-checking of a recently introduced extension of SDF graphs known as Scenario-Aware Dataflow (SADF) [23] is done in [24] utilising the CADP tool suite [7] by the application of Interactive Markov Chains (IMC). Nevertheless, it does not investigate the calculation of throughput nor does it consider multiprocessor platforms.

To the best of our knowledge, there are no papers that present a technique of finding a $maximal$ throughput without translating SDF graphs to equivalent HSDF graphs on a given number of processors, both in homogeneous and heterogeneous systems.

III. Synchronous Dataflow

In this section, the formal definitions and semantics of SDF graphs are introduced.

A. SDF Graphs

In typical streaming applications, there is a set of tasks to be executed in a certain order. An important part of these applications is a set of periodically executing tasks which consume and produce fixed amounts of data. An SDF graph is a directed, connected graph in which these tasks are represented by *actors*, data communicated is represented by $tokens$ and the $edges$ transport tokens between actors. Each edge is connected to precisely one producer and precisely one consumer. The execution of an actor is known as an $(actor)$ *firing* and the number of tokens consumed or produced onto an edge as a result of a firing is referred to as $consumption$ and $production\ rates$ respectively. In the origial definition [16], each actor takes unit time to complete its firing. However, there is a natural extension by which a certain execution time is associated with each actor [19].

Example 1. Figure 1 shows an SDF graph with three actors *u, v, w*. Arrows between the actors depict the edges which hold tokens (dots). The execution time of the actors is represented by a number inside the actor nodes. The numbers near the source and destination of each edge are the rates.

An SDF graph is defined in the following.

Definition 1. An SDF Graph is a tuple $G = (A, D, \mathsf{Tok}_0, \tau)$ where:

- A is a finite set of actors,
- D is a finite set of dependency edges $D \subseteq A^2 \times \mathbb{N}^2$,
- $\mathsf{Tok}_0 : D \to \mathbb{N}$ denotes initial tokens in each edge and

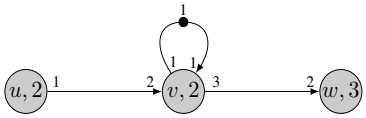


Fig. 1: SDF Graph (taken from [5])

- $\tau : A \to \mathbb{N}_{\geq 1}$ assigns an execution time to each actor.

For an SDF graph G with actors $A = \{a, b\}$, a dependency edge $d = (a, b, p, q)$ denotes a data dependency of actor b on actor a. The firing of actor a results in the production of p tokens on edge d. If the number of tokens on edge d is greater than or equal to q, actor b can execute, and as a result, it consumes q tokens from edge d.

Definition 2. The sets of input edges *In(a)* and output edges *Out(a)* of an actor $a \in A$ are defined as

$$In(a) = \{(a', a, p, q) \in D | a' \in A \wedge p, q \in \mathbb{N}\}$$

$$Out(a) = \{(a, b, p, q) \in D | b \in A \wedge p, q \in \mathbb{N}\}$$

Informally, for all actors $a \in A$ and dependency edges $d \in D$, if the number of tokens on every input edge $(a'_i, a, p_i, q_i) \in In(a)$ is greater than or equal to q_i, actor a fires and removes q_i tokens from every $In(a)$. The firing takes place for $\tau(a)$ time units and it ends by producing p_i tokens on all $(a, b_i, p_i, q_i) \in Out(a)$. For example, actor *v* in Figure 1 takes in 2 tokens from the edge *u-v* and 1 token from the edge *v-v*, fires for 2 time units and produces 3 tokens on the edge *v-w* and 1 token on the edge *v-v*.

Definition 3. The consumption rate $CR(a, b, p, q)$ and production rate $PR(a, b, p, q)$ of an edge $(a, b, p, q) \in D$ are defined as

$$CR(a, b, p, q) = q$$

$$PR(a, b, p, q) = p$$

B. Semantics

The dynamic behaviour of an SDF graph can be best understood if we define it in terms of a labelled transition system. For this purpose, we need to define the notions of state, transition and execution [9] [21].

Definition 4. The *state* of an SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ is a pair (ρ, υ). Here, $\rho : D \to \mathbb{N}$ associates with each edge the number of tokens it currently holds and $\upsilon : A \to \mathbb{N}^{\mathbb{N}}$ records for each firing of actor $a \in A$ that occurred in the past, the remaining execution time. Thus, $\upsilon(a)(k)$ denotes the number of firings of $a \in A$ that complete in exactly k time units. The initial state of an SDF graph is defined as $(\mathsf{Tok}_0, \{(a, \emptyset) | a \in A\})$ where $\emptyset$ denotes an empty multiset.

By introducing the concept of multiset of numbers for actors, it is possible to have multiple simultaneous firings of same actor also known as $auto\text{-}concurrency$. An edge $(a, b, p, q) \in D$ in an SDF graph is called a $self\text{-}loop$ if *a=b*. Auto-concurrency of any actor can be trivially restrained by adding self-loops with initial tokens equal to the desired degree of auto-concurrency. Suppose that the state vector of the SDF graph in Figure 1 is (ρ, υ) where ρ corresponds to edges *u-v, v-w, v-v* respectively and

υ represents the multisets for actor *u, v* and *w* respectively. The initial state of the SDF graph in Figure 1 is $((0,0,1),(\emptyset,\emptyset,\emptyset))$.

The transitions are of three forms i.e. the start transition representing the start of actor firing, the end transition representing the end of actor firing and discrete clock ticks representing the progress of time.

Definition 5. A transition of an SDF graph $(A, D, \mathsf{Tok_0}, \tau)$ from state (ρ_1, υ_1) to (ρ_2, υ_2) is denoted as $(\rho_1, \upsilon_1) \xrightarrow{\kappa} (\rho_2, \upsilon_2)$ and label κ is defined as $\kappa \in (A \times \{start, end\}) \cup \{tick\}$ and corresponds to the type of transition.

- Label $\kappa = (a, start)$ denotes the starting of a firing by an actor $a \in A$. For all $a \in A$ and $d \in In(a)$, this transition results in,

$$\rho_2(d) = \begin{cases} \rho_1(d) - CR(d), & \text{if } \rho_1(d) \geq CR(d) \\ & \forall a \in A \text{ and } \forall d \in In(a) \\ \rho_1(d), & \text{otherwise} \\ & \forall a \in A \text{ and } \forall d \in In(a) \end{cases} \tag{1}$$

$$\upsilon_2(a) = \begin{cases} \upsilon_1(a) \uplus \tau(a), & \text{if } \rho_1(d) \geq CR(d) \\ & \forall a \in A \text{ and } \forall d \in In(a) \\ \upsilon_1(a), & \text{otherwise.} \\ & \forall a \in A \text{ and } \forall d \in In(a) \end{cases} \tag{2}$$

  where $\uplus$ represents multiset union; that is we remove $CR(d)$ tokens and attach a's execution time $\tau(a)$ to υ_2 for all $a \in A$ and $d \in In(a)$.

- Label $\kappa = (a, end)$ denotes the ending of a firing by an actor $a \in A$. For all $a \in A$ and $d \in Out(a)$, this transition results in,

$$\rho_2(d) = \begin{cases} \rho_1(d) + PR(d), & \text{if } 0 \in \upsilon_1(a) \\ & \forall a \in A \text{ and } \forall d \in Out(a) \\ \rho_1(d), & \text{otherwise} \\ & \forall a \in A \text{ and } \forall d \in Out(a) \end{cases} \tag{3}$$

$$\upsilon_2(a) = \begin{cases} \upsilon_1(a) \backslash \{0\}, & \text{if } 0 \in \upsilon_1(a) \\ & \forall a \in A \text{ and } \forall d \in Out(a) \\ \upsilon_1(a), & \text{otherwise.} \\ & \forall a \in A \text{ and } \forall d \in Out(a) \end{cases} \tag{4}$$

  where $\backslash$ represents multiset difference. This transition produces the specified number of tokens on the outgoing edge of a and removes from υ_1 one occurrence of a with remaining executing time 0 for all $a \in A$ and $d \in Out(a)$.

- Label $\kappa = tick$ denotes a clock tick transition. For all $a \in A$ and $d \in D$, this transition is enabled if $0 \notin \upsilon_1(a)$ and results in $\rho_2(d) = \rho_1(d)$ and $\upsilon_2 = \{(a, \upsilon_1(a) \ominus 1) | a \in A\}$ where $\upsilon_1(a) \ominus 1$ denotes a multiset of elements of $\upsilon_1(a)$ decreased by one. This transition decreases by 1 the remaining execution time for all actor occurrences.

C. Scheduling

A *schedule* of an SDF graph is a firing sequence of actors to meet certain design objectives. A key aspect in SDF graphs is to find schedules with certain optimality properties, e.g. maximal throughput or the minimum number of processors required.

Definition 6. An execution of an SDF graph $(A, D, \mathsf{Tok_0}, \tau)$ is defined as an infinite sequence of states and transitions $s_0 \xrightarrow{\kappa_0} s_1 \xrightarrow{\kappa_1} \ldots$ starting from initial state of SDF graph such that $\forall n \geq 0, s_n \xrightarrow{\kappa_n} s_{n+1}$.

SDF graphs may end up in a deadlock or with an unbounded accumulation of tokens in a certain edge due to inappropriate consumption and production rates in case of non-terminating programs.

Definition 7. An SDF graph experiences a deadlock if and only if its execution has a state (ρ, υ) in which $\forall a \in A$ and $\exists d \in In(a)$ such that $\rho(d) \ngeq CR(d)$ and $\upsilon(a) = \emptyset$.

Note that deadlocked executions are infinite, as time can always progress. To avoid these effects, there is a property termed *consistency* which must hold [14] (although it does not guarantee deadlock freedom [9]). Consistency is defined as follows:

Definition 8. A repetition vector of an SDF graph $(A, D, \mathsf{Tok_0}, \tau)$ is a function $\gamma : A \to \mathbb{N}_0$ such that for every edge $(a, b, p, q) \in D$ from $a \in A$ to $b \in A$, the following equality holds.

$$p.\gamma(a) = q.\gamma(b)$$

Repetition vector γ is termed non-trivial if and only if $\forall a \in A, \gamma(a) > 0$. An SDF graph is *consistent* if it has a non-trivial repetition vector.

A repetition vector determines how often each actor must fire with respect to the other actors without a change in the token distribution. If each actor of an SDF graph is invoked according to its repetition vector in a schedule, the number of tokens on each edge is the same after the schedule is executed as before. Such a schedule is termed a periodic schedule. The repetition vector can be written in the form of matrix-vector [15] as:

$$\Gamma\gamma = \mathbf{0}, \tag{5}$$

where Γ is termed the *topology matrix* of an SDF graph and $\mathbf{0}$ is a null vector. The rows and columns of Γ are indexed by the edges and actors in an SDF graph respectively. For every edge $(a, b, p, q) \in D$ from $a \in A$ to $b \in A$, the entries of the topology matrix are defined as:

$$\Gamma((a, b, p, q), a') = \begin{cases} p, & \text{if } a' = a \\ -q, & \text{if } a' = b \\ 0, & otherwise. \end{cases} \tag{6}$$

A self-loop rules out the possibility of a repetition vector if $p \neq q$ as it contradicts equation 6; otherwise it does not have any effect on the existence of a repetition vector and is therefore not added to the topology matrix.

Lemma III.1. For γ in the equation 5 to be a vector containing only positive integers, the rank of Γ must not be full.

Proof: If the rank of Γ is full, it implies that Γ is invertible. Then we can write the equation 5 as,

$$\Gamma^{-1}\Gamma\gamma = \Gamma^{-1}\mathbf{0}$$

$$I\gamma = \mathbf{0}$$

where I is an identity matrix. The above equation is valid only if γ is a vector with all entries equal to 0, which clearly is a contradiction. ■

The rank of γ of an SDF graph is always equal to n or $n-1$ where n is the number of actors [15]. Therefore, it is necessary for Γ to have a rank $n-1$ for a repetition vector to exist [15].

Theorem III.2. An SDF graph with n actors has a periodic schedule if and only if its topology matrix Γ has a rank $n-1$. Furthermore, if its topology matrix has a rank $n-1$, then there exists a unique smallest integer solution γ to the equation $\Gamma\gamma = 0$ and all entries in the vector γ are coprime.

If Γ has a rank $n-1$, we obtain the following facts by applying linear algebra [15]:

Fact III.3. There exists a vector $\gamma \neq 0$ such that $\Gamma\gamma = 0$.

Fact III.4. If $\Gamma\gamma = 0$ then $\Gamma(K\gamma) = 0$ for any constant K.

Fact III.5. If $\Gamma\gamma_1 = 0$ and $\Gamma\gamma_2 = 0$ then there exists a scalar constant K such that $\gamma_1 = K\gamma_2$.

Clearly, an SDF graph is consistent only if its topology matrix has a rank = $n-1$ where n is the number of actors. In the remaining paper, we always assume consistency.

Definition 9. Let us assume that an SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ has a repetition vector γ. An iteration is a set of actor firings such that for each $a \in A$, the set contains $\gamma(a)$ firings of a.

For the SDF graph in Figure 1, the topology matrix is given by:

$$\Gamma = \begin{pmatrix} 1 & -2 & 0 \\ 0 & 3 & -2 \end{pmatrix}$$

As we can see that the topology matrix Γ is equal to two linear independent rows, the positive integer solution i.e. repetition vector γ exists and is equal to $\langle 4, 2, 3\rangle$. This shows that the graph is consistent and graph iteration consists of 4 firings of actor *u*, 2 firings of actor *v* and 3 firings of actor *w*.

D. Modelling Finite Resources

An SDF graph typically only models an application. When mapping an application onto a hardware platform, the chosen platform imposes an extra set of constraints, which we need to take into account. Communication between actors in an SDF graph requires buffer storage capacity. Minimising buffer capacity is an important factor to improve energy costs [8]. We therefore define an edge capacity function, which yields the maximum number of tokens that can be stored on an edge. The edge capacity function also help to make an SDF graph strongly connected.

Definition 10. The *edge capacity* of an SDF graph *G* is a function $\sigma : D \to \mathbb{N}_0$ that assigns to each edge $d \in D$ the maximum number of tokens it can hold.

The capacity of an edge $(a, b, p, q) \in D$ is modelled in an SDF graph by adding an edge $(b_\sigma, a_\sigma, q_\sigma, p_\sigma) \in D$ with $CR(a, b, p, q) = PR(b_\sigma, a_\sigma, q_\sigma, p_\sigma)$ and $PR(a, b, p, q) = CR(b_\sigma, a_\sigma, q_\sigma, p_\sigma)$ [20]. The capacity of an edge $(a, b, p, q) \in D$ is denoted by the number of initial tokens on the edge $(b_\sigma, a_\sigma, q_\sigma, p_\sigma) \in D$. The SDF graph shown in Figure 1 after adding the edge capacities is shown in Figure 2. The edge

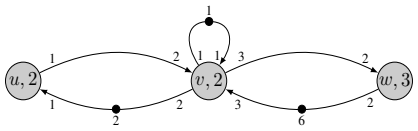


Fig. 2: SDF Graph shown in Figure 1 with edge capacities

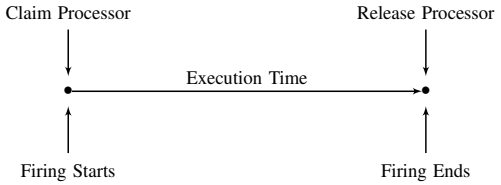


Fig. 3: Firing of an actor (taken from [26])

capacities are $\sigma(u, v, p, q) = 2$ and $\sigma(v, w, p, q) = 6$.

To avoid deadlock, an SDF graph must have a topology matrix with a rank equal to $n-1$ and enough initial tokens to execute all the firings in the repetition vector. Finding the smallest edge capacities for which the graph can be executed without the risk of a deadlock using model checking is described in [8].

Furthermore, not all actors can be mapped onto every processor, because of memory and bandwidth limitations, analogue versus digital processing capabilities, instruction set limitations, etc. as reflected in a processor application as follows:

Definition 11. A *processor application model* is a tuple (P, ζ) consisting of a finite set P of processors and a function $\zeta : P \to 2^A$ indicating which actors can be mapped to which processor.

The edge capacity function and processor application model allow us to reason about the behaviour of an application under a specific mapping. The processor is claimed by an actor at the beginning of its firing and after the execution time of the actor elapses, it finishes firing and releases the processor as shown in Figure 3.

In real-life applications, the execution time of an actor varies with the type of processor onto which it is mapped. Since we associate with each actor precisely one execution time, at the moment our model covers instances where actors are mapped onto only one type of processor in case of a heterogeneous platform.

Definition 12. A processor availability function δ on a set of processors P is given by $\delta : P \to \{0, 1\}$.

We define claiming of a processor $p \in P$ by an actor $a \in A$ at the start of its firing by $Clm : A \to (P \to \{1\})$. Similarly, releasing of a processor $p \in P$ by an actor $a \in A$ at the end of its firing is defined by $Rel : A \to (P \to \{0\})$.

Definition 13. A state of an SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ mapped on a processor application model (P, ζ) is a triple (ρ, δ, υ) [26]. Edge quantity $\rho : D \to \mathbb{N}$ associates with each edge the number of tokens present in that edge and δ associates with each processor $p \in P$ if it is available or occupied. To observe the progress of time, $\upsilon : A \to \mathbb{N}^{\mathbb{N}}$ associates a multiset of numbers representing the remaining execution times of active actor firings.

Definition 14. A transition of an SDF graph $(A, D, \mathsf{Tok}_0, \tau)$

mapped on a processor application model (P, ζ) from state $(\rho_1, \delta_1, \upsilon_1)$ to $(\rho_2, \delta_2, \upsilon_2)$ is denoted as $(\rho_1, \delta_1, \upsilon_1) \xrightarrow{\kappa} (\rho_2, \delta_2, \upsilon_2)$ and label κ is defined as $\kappa \in (A \times \{start, end\}) \cup \{tick\}$ and corresponds to the type of transition.

- Label $\kappa = (a, start)$ denotes starting of a firing by an actor $a \in A$. For all $a \in A$, $d \in In(a)$ and $p \in P$, this transition may occur if $\rho_1(d) \geq CR(d)$, $\delta_1(p) = 0$ and $a \in \zeta(p)$ and results in $\rho_2(d) = \rho_1(d) - CR(d)$, $\upsilon_2(a) = \upsilon_1(a) \uplus \tau(a)$ and $\delta_2(p) = Clm(a)(p)$. Here, $\uplus$ represents multiset union.
- Label $\kappa = (a, end)$ denotes ending of a firing by an actor $a \in A$. For all $a \in A$, $d \in Out(a)$ and $p \in P$, this transition can occur if $0 \in \upsilon_1(a)$ and results in $\rho_2(d) = \rho_1(d) + PR(d)$, $\upsilon_2(a) = \upsilon_1(a) \backslash \{0\}$ and $\delta_2(p) = Rel(a)(p)$. Here, $\backslash$ represents multiset difference.
- Label $\kappa = tick$ denotes a clock tick transition. This transition is enabled if $0 \notin \upsilon_1(a)$ for all $a \in A$. For all $a \in A$, $d \in D$ and $p \in P$, this transition results in $\rho_2(d) = \rho_1(d)$, $\delta_1(p) = \delta_2(p)$ and $\upsilon_2 = \{(a, \upsilon_1(a) \ominus 1) | a \in A\}$ where $\upsilon_1(a) \ominus 1$ denotes a multiset of elements of $\upsilon_1(a)$ decreased by one.

IV. Throughput Analysis

A. Throughput Analysis by Self-Timed Execution

The maximal throughput of an SDF graph is determined from a specific type of execution known as a *self-timed execution* [9] in which every actor fires *as soon as* it is enabled.

Definition 15. An execution is self-timed if and only if clock transitions occur when no start transitions are enabled.

Due to the deterministic behaviour of an SDF graph, the states are repeated in an execution after a certain number of firings.

Proposition IV.1. According to [9], for every consistent and strongly connected SDF graph, the state-space of a self-timed execution consists of a finite sequence of states (*transient phase*) followed by a periodic sequence repeated infinitely (*periodic phase*).

In a self-timed execution, a certain state that was visited before is revisited implying the fact that execution is then in the periodic phase. The periodic phase of an SDF graph consists of a whole number of iterations. Moreover, each actor fires according to the repetition vector in an iteration. For each actor $a \in A$ in the SDF graph, we define its corresponding entry in the repetition vector γ as $\gamma(a)$. We also define the number of iterations per period as m.

Definition 16. The throughput of an SDF graph with a processor application model is the average number of graph iterations that are executed per unit time, measured over a sufficiently long period.

The self-timed execution of the SDF graph shown in Figure 2 is explained in Figure 4. It is worth noting that after 2 simultaneous firings of actor *u* on processors p_0 and p_1, an iteration is completed every 9 time units and hence throughput is $\frac{1}{9}$. Similarly, self-timed execution in terms of the state vector

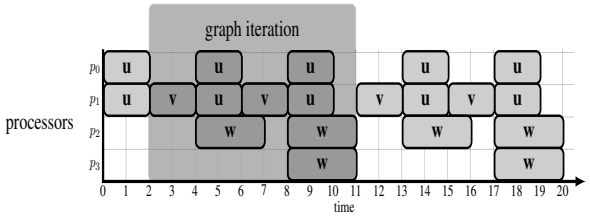


Fig. 4: Self-timed execution of SDF graph shown in Figure 2

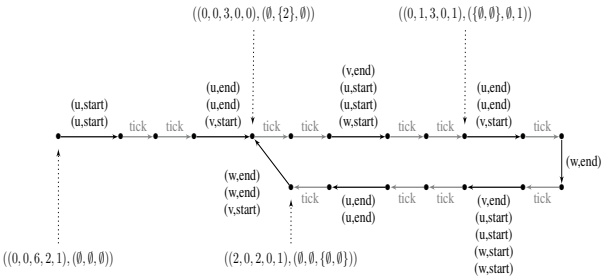


Fig. 5: Self-timed execution of our running example

(ρ, υ) of the same SDF graph is shown in Figure 5 where the edges *u-v, v-w, w-v, v-u* and *v-v* are represented by ρ respectively. Similarly, υ corresponds to the multisets for actor *u, v* and *w* respectively. We can also see that the periodic phase has a duration of 9 time units consisting of precisely one iteration. This method is implemented in the SDF^3 to calculate the throughput of SDF graphs.

B. Throughput Analysis by Fastest Execution

Let (ρ_0, υ_0) and (ρ_r, υ_r) denote the initial and recurrent states at the completion of the periodic phase respectively in a self-timed execution. For each actor $a \in A$, let f_{a_t} and f_{a_p} represents the number of times actor $a \in A$ fires in the transient phase and periodic phase respectively.

Lemma IV.2. If a periodic phase in a self-timed execution is repeated for n times, then f_{a_p} is equal to $nm\gamma(a)$.

Proof: The proof follows from the definition of self-timed execution, repetition vector and iteration. ■

The self-timed execution takes a minimum amount of time to revisit (ρ_r, υ_r) and provides the maximum throughput of an SDF graph. Therefore, we can consider it as a *fastest execution* to reach (ρ_r, υ_r) again.

Lemma IV.3. As a result of the fastest execution, let us say that the SDF graph has repeated the periodic phase n times and is in the state (ρ_r, υ_r). From here, if the SDF graph is executed in such a way that each actor $a \in A$ fires equal to $f'_{a_t} = k\gamma(a) - f_{a_t}$ for some constant k, the SDF graph reaches the initial state (ρ_0, υ_0).

Proof: Total number of firings for each actor $a \in A$ in this case are:

$$= f_{a_t} + f_{a_p} + f'_{a_t}$$
$$= f_{a_t} + nm\gamma(a) + k\gamma(a) - f_{a_t}$$
$$= (nm + k)\gamma(a)$$

From Fact III.4, $\Gamma(n\gamma) = 0$ for any constant n. ■

A necessary condition for previous lemma to hold is $f'_{a_t} \geq 0$. To reach (ρ_0, υ_0) from (ρ_r, υ_r) in the least number of firings, f'_{a_t} must be minimal. Let k_{min} denote the smallest k such that $f'_{a_t} \geq 0$ and f'_{a_t} is minimal for all actors $a \in A$.

If we assume that the part of execution from (ρ_r, υ_r) to (ρ_0, υ_0) is *fastest* also, then we can say the following.

Lemma IV.4. The fastest execution of every consistent and strongly connected SDF graph repeats the periodic phase n times if each actor $a \in A$ fires equal to $(nm + k_{min})\gamma(a)$ for some constants n and k_{min}.

Proof: Trivial for non-zero transient phase following lemma IV.2 and IV.3. If a transient phase does not exist and the SDF graph enters the periodic phase directly, then $f_{a_t} = 0$. In this case, the minimum value of k satisfying $f'_{a_t} \geq 0$ is $k_{min} = 0$. Furthermore, the total number of firings is equal to $nm\gamma(a)$ for each $a \in A$ and the periodic phase is repeated n times. ■

In section VII, we propose UPPAAL as a tool to compute the repetition vector and throughput. UPPAAL can automatically verify a number of properties, including invariant and reachability checking. An important feature in our approach is the option of generating a trace with the shortest possible accumulated time delay to reach the final state i.e. $(nm+k_{min})\gamma(a)$ for each actor $a \in A$ from the initial state (ρ_0, υ_0), termed *Fastest Trace*. UPPAAL explores the whole state-space and finds the fastest execution trace containing the periodic phase repeated n times. From the periodic phase, we determine the maximal throughput of the SDF graph.

Self-timed execution assumes there is an *unbounded* number of processors to accommodate all enabled firings of all actors at a certain time. Let P^{min} denote the finite set containing the minimum number of processors required to allow self-timed execution. From lemmas IV.3 and IV.4, we can generalise the following.

Lemma IV.5. For every consistent and strongly connected SDF graph mapped on a processor application model (P, ζ) in such a way that $\bigcup_{\forall p \in P} \zeta(p) = A$ and $\emptyset \subset P \subseteq P^{min}$, the maximal throughput of the SDF graph is determined from the periodic phase of the fastest execution to the i^{th} multiple of the repetition vector for some constant i.

Proof: In a strongly connected and consistent SDF graph, each actor depends on the other actors in order to have a sufficient amount of tokens on its input edges to be enabled for firing. This implies a bound on the difference in the number of firings of each actor with respect to the corresponding entries in the repetition vector. The state-space of reaching the i^{th} multiple of the repetition vector for some constant i if $\emptyset \subset P \subseteq P^{min}$ could contain multiple possible executions. If we search the whole state-space and consider only the fastest execution out of all executions, we notice that it contains a periodic phase implying the maximal throughput.

The reason is that in a fastest execution, if insufficient processors are available to map all simultaneous enabled firings, some of the firings will be delayed. Delaying a certain firing does not change any dependency. Instead, successors firings would also be delayed. The constraint of having to reach the final state in the least possible time ensures that delayed firings are mapped in such a way that they cause the least delay for their successor firings to be enabled. As the number of simultaneous firings of the actors and number of tokens in any edge remains bounded, the state-space is also finite. This ensures that a certain state (ρ_r, υ_r) will be revisited eventually during the execution representing the periodic phase. We explore the whole state-space with UPPAAL and find the fastest execution trace from all possible executions. ■

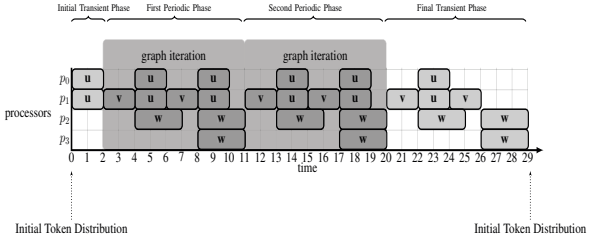


Fig. 6: Schedule using four processors

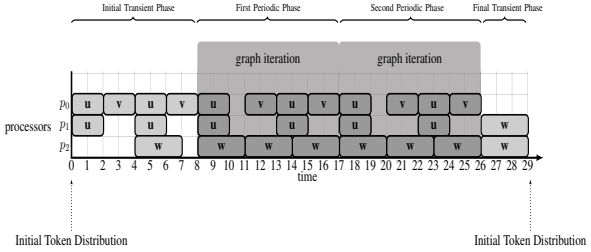


Fig. 7: Schedule using three processors

For each SDF graph, the value of k_{min} varies by altering the given number of processors and depends on how many times each actor $a \in A$ has fired during the transient phase. Therefore, the value of $nm + k_{min}$ given to UPPAAL as a final state must be high enough to ensure that f'_{a_t} is greater than 0 and the SDF graph enters the periodic phase.

Example 2. The minimum number of processors to achieve self-timed execution for the SDF graph in Figure 2 is $P^{min} = 4$. If we map the same SDF graph on 4 processors, then the fastest execution to the 3^{rd} multiple of repetition vector i.e. $3\gamma = \langle 12, 6, 9\rangle$ is shown in Figure 6. In this example, the values of n, m and k_{min} are 2, 1 and 1 respectively. Therefore, the periodic phase is repeated twice. We could determine the throughput from the periodic phase which is equal to $\frac{1}{9}$.

In the similar fashion, if we map the same SDF graph on 3 processors, the fastest execution to the 3^{rd} multiple of repetition vector is shown in Figure 7. Please note that the the value of throughput still remains $\frac{1}{9}$. In the rest of the paper, we do not analyse final transient phase as it does not affect the throughput.

V. TIMED AUTOMATA

This section recalls the basic definitions of timed automata (TA) [2], [3]. We use *B(C)* to denote the set of clock constraints for a finite set of clocks *C*. That is, *B(C)* contains all of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

Definition 17. A timed automaton $\mathcal{A}$ is a tuple (L, Act, C, E, Inv, l^0), where L is a set of *locations*; Act is a finite set of *actions*, co-actions and internal λ-actions; C

is a finite set of *clocks*; $E \subseteq L \times Act \times B(C) \times 2^C \times L$ is a set of *edges*; $Inv : L \rightarrow B(C)$ assigns an *invariant* to each location; and $l^0 \in L$ is the *initial location*.

Edges are labelled with tuples (g, α, D). Here, g is a constraint on the clocks of the timed automaton expressing when the transition can be taken; α is an action used for communication between automata; and $D \subseteq C$ is a set of clocks to be reset on the edge. For example, the edge from InUse_w to Idle in Figure 8b indicates that this edge can be taken when clock $x = 3$, its action label is end[i][w]!, and it resets no clocks. Invariants indicate when actions *have* to be taken, i.e. a timed automaton can only be in a location if all its invariants are true. For instance, the timed automaton in Figure 8b can only remain in the location InUse_w if it fulfils $x \leq 3$.

Complex TA are often built by putting together smaller component TA, using the parallel composition operator $||$. Two components in a composition synchronise on joint actions, while evolving independently on non-joint actions.

Definition 18. Let $\mathcal{A}_i = (L_i, Act_i, C_i, E_i, Inv_i, l_i^0)$, $i = 1, 2$ with $H \subseteq Act_1 \cap Act_2$ and $C_1 \cap C_2 = \varnothing$. The timed automata $\mathcal{A}_1 || \mathcal{A}_2$ is defined as,

$$(L_1 \times L_2, Act_1 \cup Act_2, C_1 \cup C_2, E, Inv_1 \wedge Inv_2, l_1^0 \times l_2^0)$$

The edge set *E* is the smallest set that contains the following transitions

- if $\alpha \in H$, $l_1 \xrightarrow{g_1:\alpha, D_1}_1 l_1'$, $l_2 \xrightarrow{g_2:\alpha, D_2}_2 l_2'$, then $\langle l_1, l_2 \rangle \xrightarrow{g_1 \wedge g_2:\alpha, D_1 \cup D_2} \langle l_1', l_2' \rangle \in E$.
- If $\alpha \notin H$, $l_1 \xrightarrow{g:\alpha, D}_1 l_1'$, $\langle l_1, l_2 \rangle \xrightarrow{g:\alpha, D} \langle l_1', l_2 \rangle$ then $l_2 \xrightarrow{g:\alpha, D}_2 l_2'$, $\langle l_1, l_2 \rangle \xrightarrow{g:\alpha, D} \langle l_1, l_2' \rangle$.

VI. Translation of SDF graphs to Timed Automata

Our framework of scheduling SDF graphs consists of separate models of an SDF graph and the processors. This method splits the scheduling problem of the SDF graphs in terms of the tasks and resources. In this section, we explain the translation of an SDF graph along with a processor application model to timed-automata with the help of Uppaal.

Given an SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ together with a processor application model (P, ζ), we generate a parallel composition of TA:

$$A_G \| Processor_1 \| \ldots \| Processor_n,$$

as shown in Figure 8. Here, the timed automaton A_G models the SDF graph as shown in Figure 8a. The TA $Processor_1, \ldots, Processor_n$ model the processors $P = \{p_1, \ldots, p_n\}$, as shown in Figure 8b. Note that the resulting timed automaton is trivially extensible in the number of processors. Thus, the translation is, at least, composable with regards to the processor application model.The underlying LTS of *G* is given by $(S, Lab, \rightarrow_G)$ where $S = (\rho, \eta)$ denotes the states, $Lab = \kappa$ denotes the labels and $\rightarrow_G \subseteq S \times Lab \times S$ denotes the edges. A_G is defined as,

$$A_G = (L, Act, C, E, Inv, \mathsf{Initial})$$

where $L = l_0 = \{\mathsf{Initial}\}$ is the only location in our SDF graph model. The action set $Act = \{\mathsf{fire!}, \mathsf{end?}\}$ contains two parameterised actions i.e. fire! (exclamation mark signifies a sending operation) and end? (question mark signifies a receiving operation) to synchronise with the TA $Processor_1, \ldots, Processor_n$.

For each processor $p_i \in P$ and actor $a \in A$, fire[i][a] represents the start of the execution of actor a on a processor p_i, and end[i][a] represents its ending. The action fire[i][a] is enabled if the incoming buffers of actor a have sufficient tokens.

We do not have any clocks and invariants in A_G. Therefore, *Inv:* $L \rightarrow$ *B(C)* and *Inv*$(l^0) = true$. For each $a \in A$ and all $d \in In(a)$, E contains two edges such that:

- Initial $\xrightarrow{\rho(d) \geq CR(d):\mathsf{fire[i][a]!}, \varnothing}$ Initial and
- Initial $\xrightarrow{true:\mathsf{end[i][a]?}, \varnothing}$ Initial.

Here, $\rho(d) \geq CR(d)$ refers to a *guard* and it signifies that tokens on all input edges $d \in In(a)$ of an actor $a \in A$ must be greater than or equal to their consumption rate in order to take the action fire!. As a result of taking the action fire!, tokens on all input edges $d \in In(a)$ of an actor $a \in A$ are subtracted i.e. $\rho(d) = \rho(d) - CR(d)$. Similarly, by taking the action end?, actor firing is completed and tokens are produced on all output edges $d \in Out(a)$ of an actor $a \in A$ i.e. $\rho(d) = \rho(d) + PR(d)$.

A_G contains a number of variables: for each edge from actors $a \in A$ to $b \in B$, an integer variable buff_a2b containing the number of tokens in the buffer from $a \in A$ to $b \in B$; counter_a, which counts how many times actor $a \in A$ has fired; and a boolean flag_act, which is initially 1, and set to 0 as soon as any actor fires. Initially, counter_a $= 0$ and buff_a2b contains the number of tokens in the initial distribution of G.

Taking the action fire[i][a] consumes, for each actor $a \in A$ and input edge $(b, a, p, q) \in In(a)$ in G, the q tokens from the buffer buff_b2a, and is carried out by the function consume(buff_b2a, q). The action end[i][a] adds, for each actor $a \in A$ and output edge $(a, b, p, q) \in Out(a)$ in G, the p tokens on the buffer buff_a2b by carrying out the function produce(buff_a2b, p). Finally, we note that the edges are parameterised in processor id's but not in actors. This is because each edge can contain only one parameter. Since the translation is defined by induction on the structure of SDF graphs, it is also composable in the (software) application.

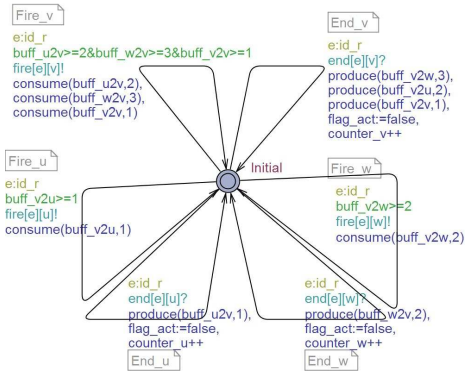
Likewise, processor TA $Processor_1, \ldots, Processor_n$ are defined as for all $1 \geq i \geq n$:

$$Processor_i = (L_i, Act_i, C_i, E_i, Inv_i, l_i^0)$$

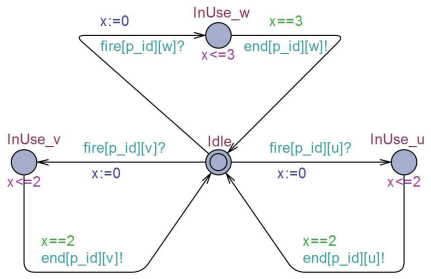
where $l_i^0 =$ Idle is an initial location and $C_i = \{x_i\}$ is a set of clocks. We do not have any invariant associated to the initial location and therefore, $Inv_i(l_i^0) = true$. For each $a \in \zeta(p_i)$, there is a set of locations $L = \{\mathsf{InUse\_a}\}$ indicating that processor $p_i \in P$ is currently used by actor $a \in A$. Furthermore, each location InUse_a is equipped with an invariant $Inv_i(\mathsf{InUse\_a}) \leq \tau(\mathsf{a})$ enforcing the system to stay in InUse_a for exactly the execution time $\tau(a)$. The action set $Act_i = \{\mathsf{fire?}, \mathsf{end!}\}$ contains for each $a \in \zeta(p_i)$, two parameterised actions fire? and end!. All actions in Act_i synchronise with A_G. For each $p_i \in P$ and $a \in \zeta(p_i)$, there are two edges,

- Idle $\xrightarrow{true:\mathsf{fire[i][a]?}, \{x_i\}}$ InUse_a where $\{x_i\}$ means clock x_i is set to zero and
- InUse_a $\xrightarrow{x_i = \tau(a):\mathsf{end[i][a]!}, \varnothing}$ Initial where $x_i = \tau(a)$ is a *guard*.

The action fire[i][a] is enabled in the initial state and leads to the location InUse_a. Thus, fire[i][a] "claims" the processor $p_i \in$



(a) UPPAAL model A_G for actors u, v, w



(b) UPPAAL model $Processor_i$ for actors u, v, w

Fig. 8: UPPAAL editor showing SDF graph and Processor

P, so that any other firing cannot run on $p_i \in P$ before the current firing of $a \in A$ is finished. As each location InUse_a has an invariant $Inv_i(\text{InUse\_a}) \leq \tau(\text{a})$, the automaton can stay in InUse_a for exactly the execution time $\tau(a)$. If $x = \tau(a)$, the system has to leave InUse_a by taking the end[i][a] action. In this way, A_G is notified that the execution of $a \in A$ has ended, so that A_G updates the buffers and other variables. Note that $Processor_i$ contains exactly one clock x_i; since clocks in UPPAAL are local we can abbreviate x_i by x. A separate clock variable records the overall time progress. For more details on translation from SDF graphs to TA and analysis in UPPAAL, we refer to [1].

VII. Analysis of SDF graphs with UPPAAL

Following Theorem III.2, starting from the initial token distribution of an SDF graph, we ask UPPAAL to find a trace which leads us to the initial token distribution again in the least possible time. We have a boolean variable `flag_act` with an initial value `true` in our UPPAAL model. As soon as the UPPAAL model starts executing, the value of `flag_act` changes to false. In a nutshell, the purpose of `flag_act` is not to give the initial state as a result and to force the model to start executing. We also associate a counter with each actor. By checking the values of counters after the query gives us a trace, we determine how many times each actor has fired to reach the target state (initial token distribution) which gives us the *repetition vector*.

As we know the initial token distribution of the SDF graph in Figure 2, selecting `Fastest` trace and verifying the following query in UPPAAL generates a trace by which we determine the repetition vector i.e. $\langle 4, 2, 3 \rangle$.

E<>(buff_u2v==0&buff_v2w==0&buff_v2u==2&buff_-w2v==6&buff_v2v==1&flag_act==false)

The repetition vector γ found in the previous step is an input to find *maximal throughput*. Following lemma IV.5, we find the fastest trace to $nm + k_{min}$-multiple of the repetition vector.

We find out the throughput of SDF graph shown in Figure 2 using $nm + k_{min} = 3^{rd}$ multiple of the repetition vector i.e. $\langle 12, 6, 9 \rangle$ by verifying the following query.

E<>(counter_u==12&counter_v==6&counter_w==9)

Figure 6 shows the schedule build from the generated trace when the SDF graph in Figure 2 is mapped on 4 processors.

Similarly, we can detect the presence or absence of *deadlocks* in an SDF graph by checking "A[] not deadlock". Please note that all counters must be removed to verify the absence of deadlocks.

VIII. Case studies

This section presents the results of the experiments in various case studies. We have used a bipartite graph with buffer capacities [8] in Figure 9, a MPEG-4 decoder [24] capable of processing 5 macro blocks in Figure 10, a MP3 playback application [25] in Figure 11, an example of SDF graphs shown in Figure 12, a MP3 decoder [20] in Figure 13 and an audio echo canceller [11] in Figure 14. Table I shows repetition vector of each SDF graph and Table II shows the results of the experiments to find out the repetition vector, throughput and deadlock freedom and comparison with SDF^3. These figures are determined using an utility called *memtime*. The experiments were run on a dual-core 2.8 GHz machine with 4GB RAM. The first column displays the number of processors, and the second column represents the value of *maximal* throughput with respect to various numbers of processors. Columns 3-6 depict memory consumption (KB) and computation time (s) required by UPPAAL in generating the trace of second multiple of the repetition vector to determine throughput, and to verify deadlock freedom. The final column represents time (s) taken by SDF^3 for calculating throughput for self-timed execution. It also explains that SDF^3 only calculates the throughput of an SDF graph assuming that a sufficient number of processors to realise self-timed execution are available.

We could determine the exact number of processors required for a self-timed execution, which is 4 for the example SDF graph in Figure 1 using SDF^3. Then, we apply our approach to derive an optimal schedule on a smaller number of processors. As we can observe from Table II, even after reducing the number of processors to 3, the throughput is the same, i.e. $\frac{1}{9}$, which clearly shows that we do not always need a self-timed execution to realise maximum throughput. If we further reduce the number of processors to 2, throughput does not deteriorate significantly and decreases slightly to $\frac{1}{11}$. Thus, using model-checking, we could generate an optimal schedule in a simple manner on a given number of processors automatically, once the target state is specified in a query. We could also check deadlock freedom efficiently if a certain SDF graph is mapped on a reduced number of processors than required for a self-timed execution.

So far, we have assumed a homogeneous system in which an actor can be mapped on any processor as all processors are identical. A homogeneous system gives more freedom to decide which actor to assign to a particular processor. However, this freedom is constrained in a heterogeneous system by which

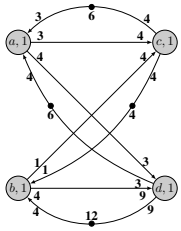


Fig. 9: Bipartite Graph [8]

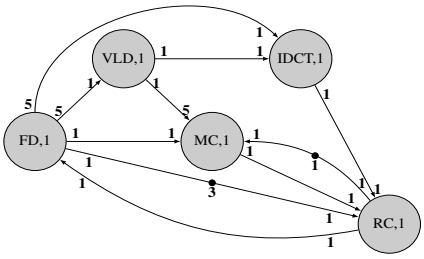


Fig. 10: MPEG-4 Decoder [24]

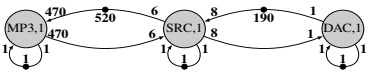


Fig. 11: MP3 Playback Application [25]

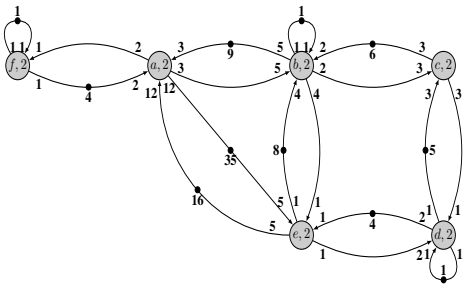


Fig. 12: Example SDF Graph

TABLE I: Repetition Vectors

Case Studies	Repetition Vector
Bipartite graph in Figure 9	[a b c d] = [12 36 9 16]
MPEG-4 Decoder in Figure 10	[FD VLD IDCT RC MC] = [1 5 5 1 1]
MP3 Playback Application in Figure 11	[MP3 SRC DAC] = [3 235 1880]
Example SDF graph in Figure 12	[a b c d e f] = [5 3 2 6 12 10]
MP3 Decoder in Figure 13	[Huffman,Req0,Req1,Redorder0,Reorder1,Stereo, Antialias0,Antialias1,Hyb Syn.0,Hyb Syn.1,Freq.Inv0, Freq.Inv1,Subb.Inv0,Subb.Inv1] = [2 1 1 1 1 1 1 1 1 1 1 1 1]
Audio Echo Canceller in Figure 14	[OUT SRC AEC ADC] = [23 23 1 23]

processors could be utilised to execute a particular actor.

In UPPAAL, we can utilise the same models described earlier in a heterogeneous system following lemma IV.5. Let us consider an SDF graph shown in Figure 1 mapped on a heterogeneous system in such a way that actor u can be mapped only on the processors p_0 and p_1, actor v can be executed only on the processor p_2, and the processor p_3 is assigned to execute actor c only. The schedule of this system is displayed in Figure 15 and the $maximal$ throughput is $\frac{1}{9}$.

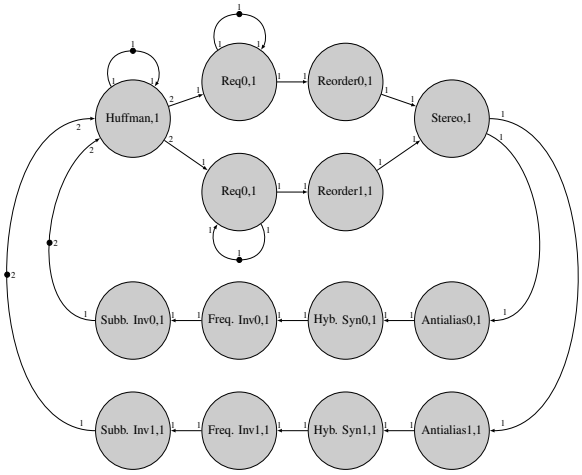


Fig. 13: MP3 Decoder [20]

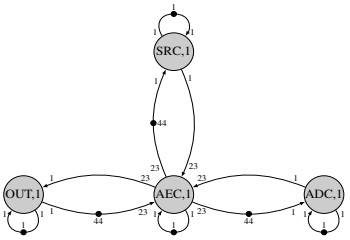


Fig. 14: Audio Echo Canceller [11]

TABLE II: Experimental Results

Proc.	Thr	Throughput		Deadlock		SDF3
		Memory	Time	Memory	Time	Time
Example SDF graph in Figure 1						
4	1/9	38144	0.3	37880	0.21	0
3	1/9	2008	0.1	2008	0.1	-
2	1/11	2008	0.1	2008	0.1	-
1	1/21	2008	0.1	2008	0.1	-
Bipartite graph in Figure 9						
4	1/42	38036	0.41	38024	0.21	0
3	1/44	37880	0.31	38008	0.2	-
2	1/51	37884	0.21	2008	0.1	-
1	1/73	2008	0.21	2008	0.1	-
MPEG-4 Decoder in Figure 10						
6	1/4	99460	259.18	41576	3.5	0
5	1/5	48960	12.04	39320	1.11	-
4	1/5	39628	0.71	38268	0.41	-
3	1/6	2008	0.1	38008	0.2	-
2	1/8	2008	0.1	2008	0.11	-
1	1/13	2008	0.1	2008	0.1	-
MP3 Playback Application in Figure 11						
2	1/1880	99176	7.25	67056	8.93	0.036002
1	1/2118	59472	1.41	47248	2.1	-
Example SDF graph in Figure 12						
5	1/24	153048	108.48	71932	36.2	0
4	1/24	63924	10.28	48600	9.66	-
3	1/28	2008	0.1	40500	1.92	-
2	1/38	2008	0.1	38284	0.3	-
1	1/76	2008	0.1	2008	0.1	-
MP3 Decoder in Figure 13						
2	1/9	38172	0.22	2008	0.1	0
1	1/15	2088	0.1	2008	0.1	-
Audio Echo Canceller in Figure 14						
4	1/23	2874728	302.97	1820852	856.36	0.004
3	1/24	484736	133.65	578080	181.36	-
2	1/25	149264	18.29	150088	26.46	-
1	1/70	55572	1.41	60856	2.82	-

IX. CONCLUSIONS AND FUTURE WORK

Despite the remarkable progress in analysis of SDF graphs, compact methods for the efficient scheduling of SDF graphs are still needed with an optimum trade-off between the maximum

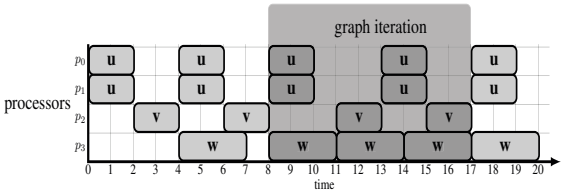


Fig. 15: Schedule in a heterogeneous system

throughput and the number of processors. By translating SDF graphs to TA, we have combined the flexibility of automata with the efficiency of SDF graphs to derive optimum schedules.

Moreover, with the help of contemporary model checkers such as UPPAAL, benefits over the range of analysable properties such as the absence of deadlocks and unboundedness, safety, liveness and reachability can also be achieved. We encountered some limitations while using UPPAAL in this context such as the state-space explosion problem for the bigger models and the inability to express complex statements such as nesting of path quantifiers.

To tackle these problems, we plan to apply multi-core LTL model checking using opaal+LTSMIN [13]. Future work also includes energy optimal reachability analysis with the help of UPPAAL CORA [18] and possibly extending the processor application model with the features such as stochastics and energy costs. Similarly, we also plan to translate a recent extension of SDF, i.e. Scenario Aware Dataflow to TA, enrich it with energy optimal reachability and mappings to Markov automata. This will allow us to achieve self energy-supporting computation in the target systems where energy generation, energy storage, and energy consumption are kept in balance over the lifetime of a system.

ACKNOWLEDGEMENT

This research is supported by the EU FP7 projects SENSATION (318490) and POLCA (610686). The authors would like to thank Bart Theelen and reviewers for their valuable comments and knowledge sharing.

REFERENCES

[1] W. Ahmad, R. de Groote, P. K. Hölzenspies, M. Stoelinga, and J. van de Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata (extended version). Technical Report TR-CTIT-13-17, University of Twente, 2014.

[2] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Seventeenth ICALP '90*, pages 322–335. Springer, 1990.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on SFM-RT '04*, LNCS, pages 200–236. Springer, 2004.

[5] E. de Groote, J. Kuper, H. J. Broersma, and G. J. M. Smit. Max-plus algebraic throughput analysis of synchronous dataflow graphs. In *38th EUROMICRO Conference on SEAA '12*, pages 29–38, 2012.

[6] M. Fakih, K. Grüttner, M. Fränzle, and A. Rettberg. Towards performance analysis of SDFGs mapped to shared-bus architectures using model-checking. In *DATE '13*, pages 1167–1172, 2013.

[7] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS '11*, Lecture Notes in Computer Science, pages 372–387. Springer Berlin Heidelberg, 2011.

[8] M. Geilen, T. Basten, and E. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *DAC '05*, pages 819–824. ACM, 2005.

[9] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. In *ACSD '06*, pages 25–34. IEEE, 2006.

[10] P. H. Hartel, T. C. Ruys, and M. C. W. Geilen. Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In *FMCAD '08*, pages 21:1–21:10. IEEE Press, 2008.

[11] J. P. Hausmans, S. J. Geuns, M. H. Wiggers, and M. J. Bekooij. Compositional temporal analysis model for incremental hard real-time system design. In *EMSOFT '12*, pages 185–194. ACM, 2012.

[12] R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[13] A. W. Laarman, M. C. Olesen, A. E. Dalsgaard, K. G. Larsen, and J. C. van de Pol. Multi-core emptiness checking of timed büchi automata using inclusion abstraction. In *CAV '13*, LNCS. Springer, July 2013.

[14] E. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, 1991.

[15] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[16] E. A. Lee and D. G. Messerschmitt. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *COMPCON '87*, pages 310–315, 1987.

[17] N. Navet and S. Merz. *Modeling and Verification of Real-time Systems*. Wiley, 2010.

[18] J. Rasmussen, K. Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In *TACAS '04*, LNCS, pages 220–235, 2004.

[19] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 1st edition, 2000.

[20] S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, 2007.

[21] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *DAC '07*, pages 777–782, New York, NY, USA, 2007. ACM.

[22] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *ACSD '06*, pages 276–278. IEEE Computer Society Press, June 2006.

[23] 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 *MEMOCODE '06*, pages 185–194, 2006.

[24] B. D. Theelen, J.-P. Katoen, and H. Wu. Model checking of scenario-aware dataflow with CADP. In *DATE '12*, pages 653–658, 2012.

[25] M. H. Wiggers. *Aperiodic multiprocessor scheduling for real-time stream processing applications*. PhD thesis, Enschede, June 2009.

[26] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs. In *ESTIMedia '09*, pages 96–105, 2009.

[27] N. E. Young, R. E. Tarjan, and J. B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21(2):205–221, 1991.