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

Waheed Ahmad[1], Mariëlle Stoelinga[1]

[1] Formal Methods and Tools Group,
University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands
w.ahmad@utwente.nl
marielle.stoelinga@ewi.utwente.nl

December 19, 2013

**Abstract**

Synchronous dataflow (SDF) graphs are a widely used formalism for modelling, analysing and realising streaming applications, both on a single processor and a multiprocessing context. Efficient schedules are necessary to obtain maximal throughput with the optimum energy consumption in such a way that the number of resources used to run these applications is kept as low a possible. This paper presents an approach of scheduling SDF graphs using a proven formalism for timed systems called timed automata (TA). TA holds a good balance between the expressiveness and tractability, and are supported by many verification tools e.g. Kronos and Uppaal. We describe an algorithm for the compositional translation of SDF graphs to TA and implementation of the translation to analyse and verify SDF graphs in state-of-the-art tool Uppaal. This approach does not require any transformation of SDF graphs to HSDF graphs and helps to find the schedules with a best compromise between number of the processors required and the throughput. It also allows quantitative model checking and verification of the user-defined properties like absence of deadlocks, safety, liveness and throughput analysis. The translation also forms the basis for future work of extending SDF graphs with the new features, e.g. stochastics, energy consumption and costs. This work also strives for bridging and extending the modelling computational formalisms towards energy aspects of self supporting computation.

## 1 Introduction

Synchronous Dataflow (SDF) graphs are well-known computational models for analysing dataflow and digital signal processing applications. Recently, they are increasingly utilised for modelling and analysing multimedia applications on a multiprocessor Systems-on-Chip (MPSoC) [16].

Current 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) [7][13][6]. Transforming SDF graph to a HSDF graph leads to a

*This research has been supported by the EU FP7 project named SENSATION.

larger graph: in the worst case, its size could be exponential as compared to the original SDF graph [19]. Another state-of-the-art method [11] calculates the throughput of SDF graphs by exploring the state-space until a periodic phase is found. However, in this method, each task is executed as soon as it is enabled and it is assumed that sufficient number of resources are available to accommodate all the enabled executions at once. On the contrary, this may not be the case in real-life applications where there is always a constraint on the number of resources.

In this paper, we propose an alternative, novel approach of modelling SDF graphs and analysing schedules using Timed Automata (TA) [3]. TA is a natural choice for modelling time-critical embedded systems to check whether 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 how long a system can stay in a certain location. TA are extensively used in the verification and model-based checking of industrial cases studies and applications [17]. Furthermore, reachability of TA is also decidable because of the abstraction using region graphs [5].

Our approach can be applied directly to SDF graphs and does not require any transformation to HSDF graphs. It also efficiently makes it possible to determine trade-off between the number of the processors and the throughput for a certain application. This will aid to a huge extent in finding the efficient schedules in terms of energy and memory consumption. Moreover in multiprocessor applications, it is also possible to build a schedule for a heterogeneous system in a sound manner. In a heterogeneous system, only specific resources can run a particular task due to their computational limitations.

Quantitative model checking and support for evaluating the user-defined properties is lacking in the existing contemporary SDF graph analysis tools e.g. SDF$^3$ tool suite[21]. With the growing use of model checking, there is a huge need to fill in this gap. In this context, state-of-the art model checker UPPAAL [4] is exploited to evaluate the user-defined properties which further adds to the benefits of TA. Results have shown that other than optimal scheduling of SDF graph, we can explore the future directions of decorating SDF graphs with the new features, i.e. stochastics and energy consumption and combining with the new extensions of TA like costs and timed games.

The outline of the paper is organized as follows: section 2 reviews the related research work done. Section 3 explains the formal sematics of SDF graphs comprehensively. Section 4 covers TA and their flavour in UPPAAL and section 5 covers the algorithm developed for translating SDF graphs to TA with an example. Section 6 focuses on the implementation of this translation in UPPAAL, determining repetition vector, throughput and deadlock freedom of the case studies and the obtained results. Section 7 draws the conclusions and outlines the future research.

## 2 Related Work

We find various formalisms for dataflow models like computational graphs [13] and SDF graphs [16]. SDF graphs are more expressive because they efficiently model and analyse the embedded dataflow applications e.g. MPEG-4 and MP3 decoders on multiprocessors. Minimising the buffer requirements in SDF graph using model checking is analysed in [9] in-depth. Throughput analysis of HSDF graphs is studied extensively in [6, 18, 25, 13, 7]. An algorithm proposed by Karp in [13] to find out maximum cycle mean(MCM) is an another efficient method of calculating the throughput. All these studies are focused on studying HSDFs and require conversion of SDF graphs to HSDF graphs as explained in [16, 25]. Throughput calculation method applicable directly on SDF graphs

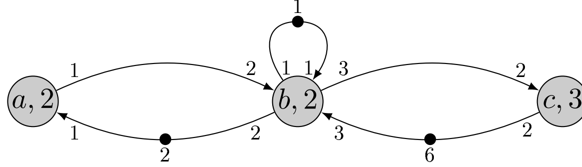



Figure 1: SDF Graph

[11] is practical only if we have infinite number of processors. On the other hand, our strategy calculates the throughput on a given number of processors.

Reference [20] presents a notion of a binding-aware SDF graph in which resources are allocated by binding the SDF graphs to a multi-processor heterogeneous system. In a binding-aware SDF graph, it is ensured that enough resources are available for each application to preserve its throughput guarantees. But these bindings impose extra constraints which results in a lower throughput. Furthermore, static order scheduling is also needed for actors within an application unlike our strategy. Model-checking of a recently introduced formalism of SDF graphs known as Scenario-Aware Dataflow (SADF) is done in [22] utilising Construction and Analysis of Distributed Processes(CADP) tool suite [8] by the application of Interactive Markov Chains (IMC). However, calculating throughput suffers from the lack of ability to assess reward-based properties in CADP.

Unlike previous approaches, our technique of conceiving SDF graph from TA point of view works directly with SDF graphs. Furthermore, it also allows model-checking and throughput analysis with desired number of processors and heterogeneous platforms.

# 3 Synchronous Dataflow

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

## 3.1 SDF Graphs

In a typical signal processing and multimedia application, there is a set of tasks to be executed in a certain order and data is transferred between them. An important part of these applications is a set of periodically executing tasks which consume and produce fixed amounts of data. In SDF graph, these tasks are represented by *actors* and data communicated is represented by *tokens*. Tokens are communicated on *edges* between actors. 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. By 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 to each actor.

**Example 1.** Figure 1 [7] shows a SDF graph with three actors *a, b, c*. Arrows between the actors depict the edges and the black dots on them represent the initial tokens. The execution time of the actors is represented by a number inside the actor nodes and associated with the source and destination of each the edge are the rates.

A SDF graph is defined as,

**Definition 1.** *A 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* $D \subseteq A^2 \times \mathbb{N}^2$,
- $\mathsf{Tok}_0 : D \rightarrow \mathbb{N}_0$ *denotes initial tokens in each edge and*
- $\tau : A \rightarrow \mathbb{N}$ *assigns an execution time to each actor.*

*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* $d$ *are greater than* $q$*, actor* $b$ *can execute and as a result, it consumes* $q$ *tokens from edge* $d$*.*

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

$$In(a) = \{(a_0, a, p, q) \in D | a_0 \in A\}$$

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

Formally, if the number of tokens on all input edges are greater than $q$, actor $a$ fires and removes $q$ tokens from all $(a_0, a, p, q) \in In(a)$. The firing takes place for $\tau$ time units and it ends in producing $p$ tokens on all $(a, b, p, q) \in Out(a)$. For example, actor $a$ in Figure 1 takes in one token from the edge $b$-$a$, continues its firing for two time units resulting in producing one token on the edge $a$-$b$.

**Definition 3.** *The consumption rate* $CR(a, b, p, q)$ *and production rate* $PR(a, b, p, q)$ *of an edge* $(a, b, p, q)$ *is defined as*

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

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

The processor application model defined below expresses a processor platform on which actors can be mapped and executed. In real-time applications, some actors cannot be mapped onto the certain processors due to memory and bandwidth limitations. Therefore, a processor application model needs information about the resource requirements of actors and determines the set of actors which can be bound onto particular processors.

**Definition 4.** *A processor application model is a tuple* $(P, \varsigma)$ *consisting of a finite set* $P$ *of processors* $P = \{P_1 \ldots P_n\}$ *and a function* $\varsigma \subseteq P \times A$ *showing actors which can be mapped on each processor.*

The processor is claimed by an actor in the beginning of its firing and after execution time of the actor elapses, it finishes firing and releases the processor as shown in Figure 2 [24].

### 3.2 Semantics

The dynamic behaviour of a SDF graph can be best understood if we define it in terms of a labelled transition system. For this purpose, we need to define notion of states, transitions and execution [11][20].

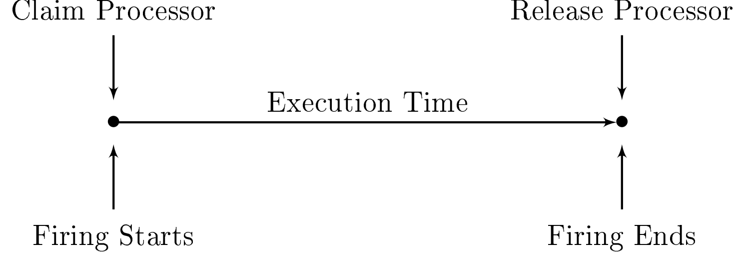



Figure 2: Firing of an actor

**Definition 5.** *The state of a SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ is a pair $(\rho, \upsilon)$ where $\rho$ associates with each edge current number of tokens present in that edge such that $\rho : D \to \mathbb{N}$. The function $\upsilon : A \to \mathbb{N}^{\mathbb{N}}$ keeps track of time progress by associating multiset of numbers representing remaining times of different firings of actor $a \in A$. The initial state of SDF graph is defined as $(\mathsf{Tok}_0, \{(a, \{\})|a \in A\})$ where $\{\}$ 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-concurrency. Auto-concurrency of any actor can be restrained by adding self-loops with initial tokens equal to desired degree of auto-concurrency. Let us suppose that the state vector of the SDF graph in Figure 1 is $(\rho, \upsilon)$ where $\rho$ corresponds to edges *a-b, b-c, c-b, b-a, b-b* respectively and $\upsilon$ explains the multisets for actor *a,b* and *c* respectively. The initial state of the SDF graph is ((0,0,6,2,1),({},{},{}).

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

**Definition 6.** *A transition of a SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ from state $(\rho_1, \upsilon_1)$ to $(\rho_2, \upsilon_2)$ is defined as $(\rho_1, \upsilon_1) \xrightarrow{\kappa} (\rho_2, \upsilon_2)$ and label $\kappa$ 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 $a$. For all $d \in In(a)$, this transition may occur if $\rho_1(d) \geq CR(d)$ and results in,*

$$\rho_2(d) = \begin{cases} \rho_1(d) - CR(d), & \text{if } \rho_1(d) \geq CR(d) \\ \rho_1(d), & \text{otherwise.} \end{cases} \tag{1}$$

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

  *where $\uplus$ represents multiset union.*

- *Label $\kappa = (a, end)$ denotes ending of a firing by an actor $a$. For all $d \in Out(a)$, this transition can occur if $0 \in \upsilon_1(a)$ and results in,*

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

$$\upsilon_2(a) = \begin{cases} \upsilon_1(a)\backslash\{0\}], & \textit{if } 0 \in \upsilon_1(a) \\ \upsilon_1(a), & \textit{otherwise.} \end{cases} \tag{4}$$

*where $\backslash$ represents multiset difference.*

- *Label $\kappa = tick$ denotes a clock tick transition. This transition is enabled if no end transition is enabled and $0 \notin \upsilon_1(a)$ for all $a \in A$. This transition 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.*

**Definition 7.** *An execution of a SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ is defined as an alternating sequence (infinite or finite) of states and transitions $s_0 \xrightarrow{\kappa_0} s_1 \xrightarrow{\kappa_1} \ldots$ starting from initial state of SDF graph such that $\forall n \geq 0, s_n \xrightarrow{\kappa_n} s_{n+1}$. An execution is maximal if and only if it is finite with none of actors enabled in the final state, or if it is infinite.*

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

**Definition 8.** *A SDF graph has a deadlock if and only if its maximal execution has a finite length. A SDF graph is deadlock free if and only if all actors fire infinitely often in an execution [10].*

If, for example, consumption rate of actor $a$ in Figure 1 is increased to 2, it would lead to a deadlock. Similarly, changing production rate of actor $a$ to 2 would cause an unbounded accumulation of tokens on the edge from actor $a$ to $b$. To avoid these effects, there is a property called consistency which must hold [15] (although it does not guarantee deadlock freedom). Consistency is defined as following.

**Definition 9.** *A repetition vector of a 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 relation exists.*

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

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

Repetition vector determines how often each actor must fire with respect to the other actors without a change in the token distribution. In the remainder, we always assume consistency.

**Definition 10.** *Let us assume that SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ has a repetition vector $\gamma$. An iteration is a set of actor firings such that for each $a \in A$, the set contains $\gamma(a)$ firings of $a$.*

By solving the balance equations $p.\gamma(a) = q.\gamma(b)$ for the SDF graph in Figure 1, we come to know that the graph is consistent and graph iteration consists of 4 firings of actor $a$, 2 firings of actor $b$ and 3 firings of actor $c$. Therefore, repetition vector is $\langle 4, 2, 3 \rangle$.

Due to the deterministic behaviour of a SDF graph, the states are repeated in an execution after a certain number of firings. According to [11], for every consistent and strongly connected

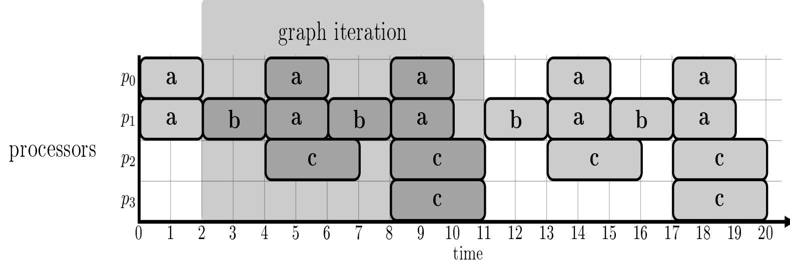



Figure 3: Self-timed schedule

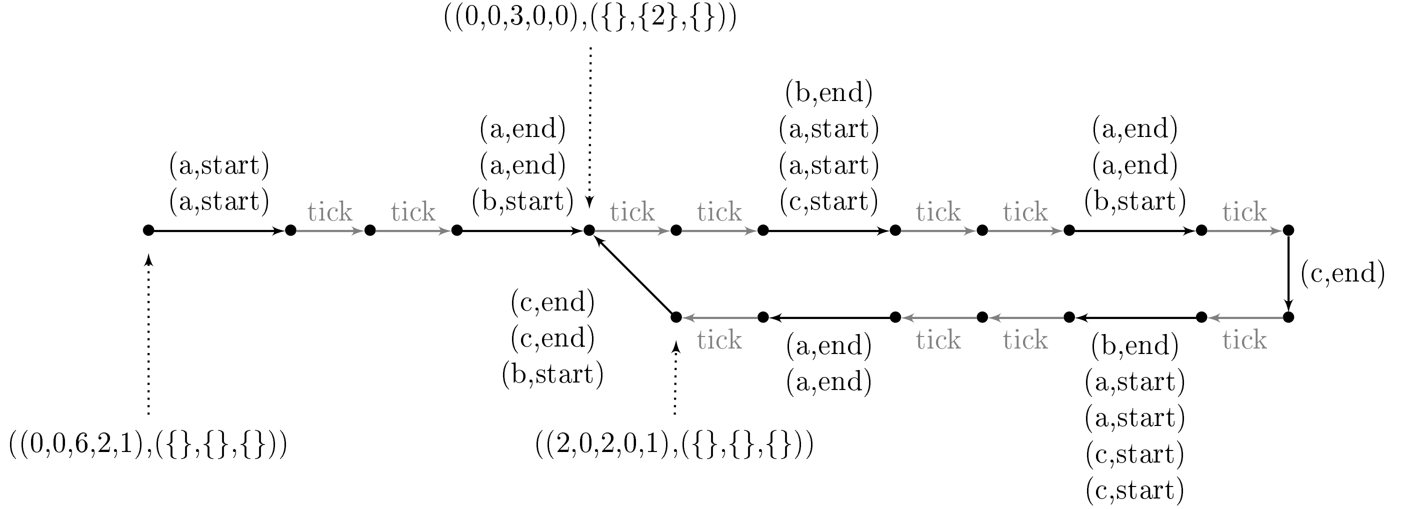



Figure 4: Self-timed execution of our running example

SDF graph, the state-space consists of finite sequences of states (called *transient phase*) followed by a periodic sequence repeated infinitely (called *periodic phase*). The periodic phase of a SDF graph consists of whole number of iterations. An iteration does not have any net effect on token distribution and the SDF graph returns back to the same state from where the periodic behaviour started. Moreover, each actor fires according to the repetition vector in an iteration.

The execution in which infinite processors are available and each actor is fired as soon as it is enabled called self-timed execution of the SDF graph in Figure 1 is explained in Figure 3. It is worth noticing that after 2 simultaneous firings of actor $a$ on processors p0 and p1, 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 $(\rho, \upsilon)$ of the same SDF graph is portrayed in Figure 4 where we can see also that its periodic phase having a duration of 9 time units consists of precisely one iteration.

# 4 Timed Automata

This section introduces the basic definitions of syntax and semantics of timed automata (TA) [2, 3]. We use the following notations: $C$ is a set of clocks and *B(C)* is a set 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 $\bowtie \in \{<, \leq, =, \geq, >\}$.

### 4.1 Definitions

**Definition 11.** *A timed automata is a tuple $TA = (L, Act, C, E, Inv, l^0)$ where $L$ is a set of locations, $Act$ is a finite set of actions, co-actions and internal $\lambda$-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 between locations with an action, a guard and a set of clocks to be reset, $Inv : L \to B(C)$ assigns invariants to locations and $l^0 \in L$ is the initial location.*

A clock valuation is a function $\eta : C \to \mathbb{R}_{\geq 0}$ from the set of clock to the non-negative real numbers. Let $\mathbb{R}^C$ be the set of all clock valuations. Edges are labelled with tuples $(g, \alpha, D)$ where $g$ is a clock constraint on the clocks of the timed automaton, $\alpha$ is an action, and $D \subseteq C$ is a set of clocks. We can interpret an edge $l \xrightarrow{g:\alpha,D} l'$ as the timed automaton can move from location $l$ to $l$' if guard $g$ is satisfied. As a result, an action $\alpha$ is performed and any clock in D is reset to zero. Let $\eta_0(x) = 0$ for all $x \in C$. We will notate $\eta$ satisfies guard $g$ by writing $\eta \models g$. Similarly, $\eta$ satisfies $I(l)$ is written as $\eta \models Inv(L)$. The semantics of TA are defined below.

**Definition 12.** *Let $(L, Act, C, E, Inv_i, l^0)$ be a timed automaton. The semantics of TA is defined as a labelled transition system $\langle S, s_0, \to\rangle$ where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, \eta_0)$ and $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup Act) \times S$ is the transition relation such that,*

- $(l, \eta) \xrightarrow{d} (l, \eta + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow \eta + d' \models Inv(l)$, *and*
- $(l, \eta) \xrightarrow{a} (l', \eta')$ *if there exists* $e = (l, a, g, r, l') \in E$ *s.t.* $\eta \models g$, $\eta' = [r \mapsto 0]\eta$, *and* $\eta' \models Inv(l')$ *where for* $d \in \mathbb{R}_{\geq 0}$, $\eta + d$ *maps each clock* $x$ *in* $C$ *to the value of* $\eta(x) + d$ *and* $[r \mapsto 0]\eta$ *denotes the clock valuation which maps each clock in* $r$ *to 0 and satisfies with* $\eta$ *over* $C \backslash r$.

Time-critical systems are often modelled as a parallel composition of TA and is denoted by a parallel composition operator $||$ parametrised with handshaking actions $H$. Actions in $H$ need to be carried out by both involved timed automata jointly.

**Definition 13.** *Let $TA_i = (L_i, Act_i, C_i, E_i, Inv_i, l_i^0)$, $i = 1, 2$ with $H \subseteq Act1 \cap Act2$ and $C_1 \cap C_2 = \varnothing$. The timed automata $TA_1 || TA_2$ is defined as,*

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

*The transition edge $E$ is defined per following rules,*

- *for* $\alpha \in H$ :

$$\frac{l_1 \xrightarrow{g_1:\alpha,D_1}_1 l_1' \wedge l_2 \xrightarrow{g_2:\alpha,D_2}_2 l_2'}{\langle l_1, l_2\rangle \xrightarrow{g_1 \wedge g_2:\alpha,D_1 \cup D_2} \langle l_1', l_2'\rangle}$$

- *for* $\alpha \notin H$ :

$$\frac{l_1 \xrightarrow{g:\alpha,D}_1 l_1'}{\langle l_1, l_2\rangle \xrightarrow{g:\alpha,D} \langle l_1', l_2\rangle} \quad and \quad \frac{l_2 \xrightarrow{g:\alpha,D}_2 l_2'}{\langle l_1, l_2\rangle \xrightarrow{g:\alpha,D} \langle l_1, l_2'\rangle}$$

Figure 5 shows an example of a timed automaton of a lamp and an user. The timed automaton of a lamp has three locations i.e. *off*, *dim* and *full*. If the user presses a switch once and synchronises with `press`?, then the lamp is on and emits dim light. The user has to press switch again to to switch off the lamp. But if full light is required, the switch must be pressed twice rapidly. The clock $y$ is used to detect if user is fast ($y$<5) or slow ($y$ ≥5).

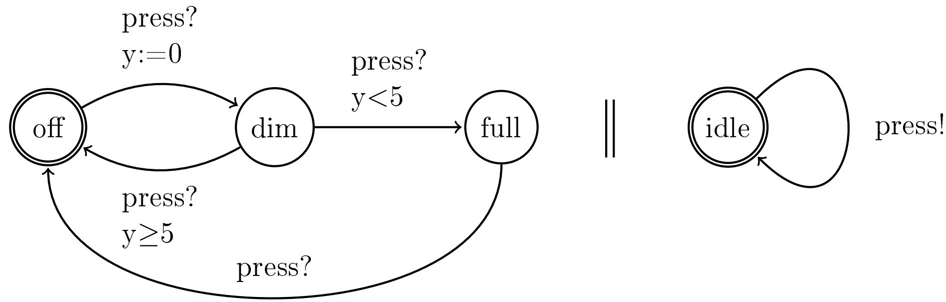



Figure 5: Timed automaton of a lamp and an user

### 4.2 Timed Automata in UPPAAL

This subsection explains the related features extended to TA by Uppaal modelling language.

A **system model** in Uppaal consists of a network of processes. The description of a model has three parts i.e. global and local declarations, automata templates and system definition.

**Declarations** are either local or global and may contain declarations of clocks,arrays, bounded integers, channels, arrays, records and types.

**Templates** automata are defined with the local declarations and a set of parameters of any type e.g. *int, chan*. A template is instantiated in system definition.

In the **system definition**, whole system model is defined in terms of one or more concurrent processes, local and global variables and channels.

Automata synchronise on *channels*. **Binary** channels model binary and blocking synchronisation and channels are declared as `chan c`. An edge labelled as `c!` denotes a sender and synchronises with another edge labelled as `c?` representing a receiver.

**Broadcasting** channels model asymmetric one-to-many synchronisation and are declared as `broadcast chan c`. In an broadcast channel, one sender `c!` can synchronise with an arbitrary number of receivers `c?`.

**Arrays** are permitted for clocks, channels, integer variables and constants. They are defined by adding a size to the variable name. For instance, `int i[4];`, `chan M[4];`, `clock y[2];` and `int x[3,5] a[7];`

**Initialisers** are used to initialise the integer variables and arrays comprising of integer variables. For example, `int i=3;` and `int i[3]={1,2,3};`

**User defined functions** are defined either globally or locally to the templates. Local functions can access the template parameters.

Expressions in Uppaal range over clocks and variables and may have the following labels. All of these expressions occur during taking an edge except *invariant* which is associated to the *locations*.

**Select** label is called in contains a comma separated list of *name : type* expressions where *name* is a variable name and *type* is a defined type.

**Guards** are side-effect free expressions on edges and evaluates to a boolean. Only clocks, integer variables and constants are referenced. Guards over clocks are essentially conjunctions.

A **synchronisation** label is of a form `Expression!` or `Expression?` or can be empty. A synchronisation label must be side-effect free.

An **update** is a comma-separated list of expressions with side-effects. Expressions in an update label must refer to clocks, integers, variables and constants only. They may also call functions.

An **invariant** is a side-effect free label and must refer to clocks, integers, variables and constants only. An invariant is a conjunction of conditions of a form `x<e` or `x<=e` where `x` is a clock and `e` evaluates to an variable.

UPPAAL toolkit has three tabs i.e. the editor, the simulator and the verifier. The key idea is that the user models a system graphically in the editor, simulates it to check its behaviour and verify it in the verifier against the set of queries.

# 5 Translation of SDF graph to UPPAAL

Total framework of scheduling SDF graphs consists of separate models of a SDF graph and the processors. This method bisects the scheduling problem of SDF graphs in terms of the tasks and resources. In this section, we will explain translation algorithm of SDF graph to timed-automata with the help of a naive representation of SDF graph model and a processor model. These naive models will help us to model in UPPAAL in the next section.

We associate to each SDF graph $G = (A, D, \mathsf{Tok}_0, \tau)$ a parallel composition of a TA

$$A_G || Processor_1 \dots Processor_n.$$

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 $\rightarrow_G \subseteq S \times Lab \times S$ depicts the edges. Here TA $A_G$ models the SDF graph and TAs $Processor_1 \dots Processor_n$ model the processors $\{P_1 \dots P_n\}$. $A_G$ is defined as

$$A_G = (L, Act, C, E, Inv, l^0)$$

where $L = l^0 = \{Initial\}$ is the only location in our SDF graph model, $Act = \{req!, fire?\}$ is a set of actions and is used to synchronise $A_G$ and $Processor_1 \dots Processor_n$. We do not have any invariants in $A_G$. Therefore, *Inv:* $L \rightarrow B(C)$ and $Inv(l^0) = true$. For all $a \in A$ and $d \in In(a)$, we have a set of edges $E = \{\text{REQ}, \text{FIRE}\}$ such that $\text{REQ} = Initial \xrightarrow{\rho(d) \geq CR(d):req!,\varnothing} Initial$ and $\text{FIRE} = Initial \xrightarrow{true:fire?,\varnothing} Initial$. $\rho(d) \geq CR(d)$ refers to a *guard* and it signifies that tokens on all input edges of an actor $a$ must be greater than or equal to their consumption rate in order to take the edge REQ. As a result of taking edge REQ, tokens on all input edges $d \in In(a)$ are subtracted by calling $\rho(d) = \rho(d) - CR(d)$ in the field *update* of UPPAAL. Similarly tokens are produced on all

**Input**: A SDF graph $(A, D, \mathsf{Tok}_0, \tau)$ and a Processor application model $(P, \varsigma)$
**Output**: Network of UPPAAL models $A_G || Processor_1 \dots Processor_n$
**for** *SDF graph* $(A, D, \mathsf{Tok}_0, \tau)$ **do**
  create a location $Initial \in l^0$ in $A_G$;
  **for** $\forall a \in A$ **do**
    create an edge $REQ \in E$ as $Initial \xrightarrow{g1:req(resource_id,actor_id)!,\varnothing} Initial$ in $A_G$ where $g1 : \rho(d) \geq CR(d)$
    create an edge $FIRE \in E$ as $Initial \xrightarrow{true:fire(resource_id,actor_id)?,\varnothing} Initial$ in $A_G$
  **end**
**end**
**for** $1 \geq i \geq n$ **do**
  create a location $Idle \in l_i^0$ in $Processor_i$;
  allocate $x_i \in C_i$ in $Processor_i$;
  **for** $\forall a \in A$ *and* $\forall (P_i, a) \in \varsigma$ **do**
    create a location $InUse_a \in L_i$ with $Inv_i(InUse_a) \leq \tau(a)$ in $Processor_i$;
    create an edge $CLAIM_i \in E_i$ as $Idle \xrightarrow{true:req(resource_id,actor_id)?_i,\{x_i\}} InUse_a$ in $Processor_i$
    create an edge $REL_i \in E_i$ as $InUse_a \xrightarrow{g_i:fire(resource_id,actor_id)!_i,\varnothing} Idle$ in $Processor_i$ where $g_i : x_i := \tau(a)$;
  **end**
**end**

**Algorithm 1:** Algorithm for translation of SDF and Processor application models to TA

output edges $d \in Out(a)$ after completion of the firing by calling $\rho(d) = \rho(d) + PR(d)$.

Similarly processor TAs $Processor_1 \dots Processor_n$ 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 all $a \in A$ and $(P_i, a) \in \varsigma$, we define a set of locations $L_i = \{InUse_a\}$, invariants are associated to the locations equal to the execution time of the actor $a$ i.e. $Inv_i(InUse_a) \leq \tau(a)$, $Act_i = \{req?, fire!\}$ is a set of actions used for synchronisation, $E_i = \{\mathrm{REQ}, \mathrm{FIRE}\}$ is a set of edges such that $\mathrm{REQ} = Idle \xrightarrow{true:req?,\{x_i\}} InUse_a$ where clock $x_i$ is set to zero and $\mathrm{FIRE} = InUse_a \xrightarrow{x_i:=\tau(a):fire!,\varnothing} Initial$ where $x_i := \tau(a)$ is a *guard*.

The translation is given in Algorithm 1. Please note that we have used two-dimensional array of channels in the algorithm where the first index selects an processor id and the second index takes an actor id. Adopting two-dimensional array makes certain that actor fires also on same processor it has requested. We will describe the implementation of this algorithm to a SDF graph example to generate the generic naive UPPAAL models in the next subsection.

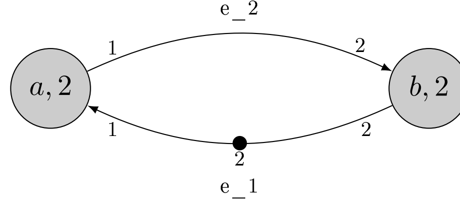



Figure 6: Example SDF Graph

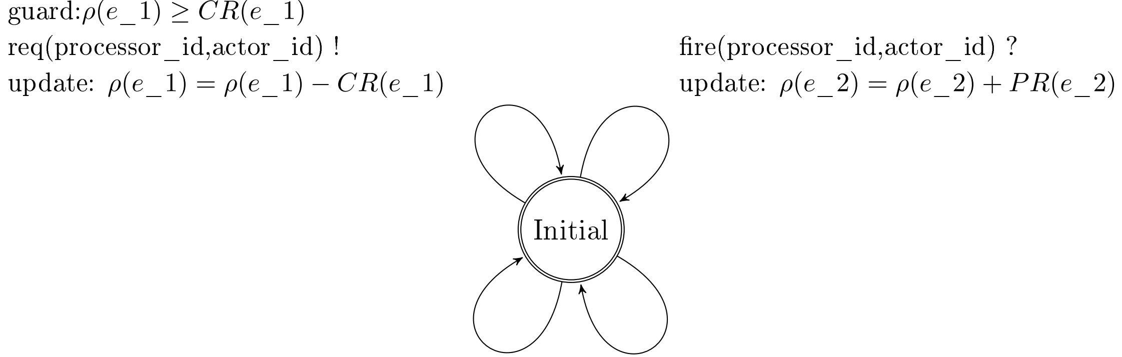



Figure 7: TA $A_G$ of a SDF graph $G$ in Figure 6

### 5.1 Example - A Naive Model

Let us consider an example portrayed in Figure 6 having two actors i.e. $a$ and $b$. Both of them have an execution time equal to 2 time units. Tokens are stored in the edges $e_1$ and $e_2$ and there are two initial tokens in the edge $e_1$. The production and consumption rate of the edge $e_1$ is 2 and 1 respectively. Similarly, the production rate of the edge $e_2$ is 1 and 2 respectively. This SDF graph is translated to a UPPAAL model using Algorithm 1 and is described below.

A SDF Graph naive model is composed of a single location called `Initial` and is depicted in Figure 7. Every actor and processor has an unique identifier `id` named as `actor_id` and `resource_-id` respectively. For each actor in SDF Graph, there are two edges in UPPAAL model. The purpose of the first edge `REQ` is to claim an empty processor. Once processor is available, the second edge `FIRE` acts to fire corresponding actor. There are integer variables `buff_b2a` and `buff_a2b` respectively for the edges $e_1$ and $e_2$ in UPPAAL model and current value of the variable exhibits current number of tokens. The initial value of the variable is equal to the initial number of tokens in that edge.

Every processor model as shown in Figure 8 has an initial location called `Idle` which represents that the processor is unoccupied. Furthermore, the processor model has a dedicated location `InUse` for each actor. This approach establishes a notion that a processor allots a limited time duration to each actor to complete its firing. Afterwards, actor has to leave the processor instantaneously. SDF graph model and processor model synchronises with each other by means of channels

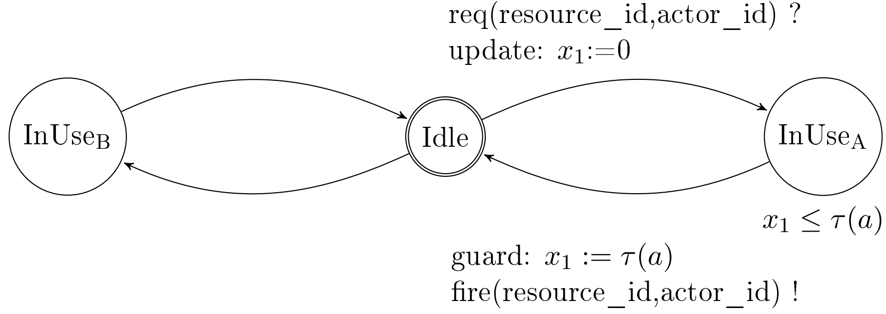



Figure 8: TA *Processor* representing a Processor

`req(resource_id,actor_id)` and `fire(resource_id,actor_id)`. A separate clock is assigned to each processor. For the sake of simplicity, edge annotations of actor $b$ are omitted in Figure 7 and Figure 8 but they are similar to edge annotations of actor $a$.

Let $TA_i = (L_i, Act_i, C_i, E_i, Inv_i, l_i^0)$ and i=1,2 respectively for SDF graph and Processor. TA semantics of SDF graph is described as following.

- $L_1 = l_1^0 = \{Initial\}$,
- $Inv_1(L_1) = true$,
- $C_1 = \varnothing$,
- $Act_1 = \{req(resource_id,\ actor_id)!, fire(resource_id, actor_id)?\}$ and
- $E_1 = \{REQ(A), FIRE(A),\ REQ(B),\ FIRE(B)\}$

TA semantics of Processor is described as following.

- $l_2^0 = \{Idle\}$,
- $L_2 = \{Idle, InUse_A, InUse_B\}$,
- $C_2 = \{x_1\}$,
- $Inv(InUse_A) \leq \tau(a)$,
- $Inv(InUse_B) \leq \tau(b)$,
- $Act_2 = \{req(resource_id,\ actor_id)?, fire(resource_id, actor_id)!\}$ and
- $E_2 = \{CLAIM(A), REL(A),\ CLAIM(B),\ REL(B)\}$

If $g \models \rho(e_1) \geq CR(e_1)$, edges $REQ(A) \in E_1$ and $CLAIM(A) \in E_2$ are taken such as,

- $Initial \xrightarrow{g:req(resource_id,actor_id)!,\varnothing} Initial$

- $Idle \xrightarrow{true{:}req(resource_id,actor_id)?,\{x_1\}} InUse_A$

As the edge REQ(A) is fired, the tokens are consumed from the incoming edges equal to their corresponding consumption rates.

If $x_1 \models Inv(InUse_A)$ and $g \models x_1 := Inv(InUse_A)$, edges $FIRE(A) \in E_1$ and $REL(A) \in E_2$ are taken such as,

- $InUse_A \xrightarrow{g{:}fire(resource_id,actor_id)!,\varnothing} Idle$
- $Initial \xrightarrow{true{:}fire(resource_id,actor_id)?,\varnothing} Initial$

As a result of the edge FIRE(A), actor produces tokens on the outgoing edges equal to their production rate and the graph keeps on executing in the same fashion.

With respect to the SDF graph in Figure 6, we can see in Figure 7 that there are two edges for actor $a$ designated as `REQ(A)` and `FIRE(A)`. Likewise, there are two edges `REQ(B)` and `FIRE(B)` for actor $b$. There are two locations $InUse_A$ and $InUse_B$ for actors $a$ and $b$ respectively in the Processor model. Lets say that *actor_id* of actor $a$ and $b$ is *aid* and *bid* respectively. We also assume that we have one processor with *resource_id* equal to p0.

As the pre-condition of firing is fulfilled in Figure 6, actor $a$ synchronises with the empty processor p0 by means of the channel *req(p0,aid)*. Subsequently, actor $a$ takes the edge `REQ(A)` and one token is removed from the edge $e_1$. As actor $a$ takes the edge `REQ(A)`, the processor moves to the location $InUse_A$ using the edge `CLAIM(A)`and the clock assigned to the processor is reset. Immediately after the execution time of actor $a$ equal to two time units finishes, the processor indicates back to actor $a$ by means of the channel *fire(p0,aid)* and finishes firing of actor $a$ by moving back to the location *Idle* by taking the edge `REL(A)`. Simultaneously, actor $a$ produces one token on the edge $e_2$ by taking the edge `FIRE(A)`.

We can produce several instances of the same processor model in Uppaal in order to enable multiple simultaneous firings of any actor. As evident from Figure 6, actor $a$ can fire twice simultaneously in the beginning. If we have two instances of template *Processor* called p0 and p1, actor $a$ can request access of both processors at the same time if they are free. Hence, there would be two parallel simultaneous firings of actor $a$ which would result in the higher throughput.

# 6 Scheduling of SDF Graphs by Model Checking

In this section, we will describe the implementation of the translation algorithm presented in the last section in Uppaal, optimal scheduling of SDF graphs and calculating the throughput. We will also explain SDF graph in Figure 1 modelled in Uppaal.

## 6.1 Implementation of SDF Graphs in UPPAAL

Let us consider the SDF graph in Figure 1 and its self-timed execution shown in Figure 3. In Uppaal, we build a separate template for the SDF graph and Processor namely `SDFG` and `Processor` respectively. As we need four processors to observe self-timed execution, we create four instances of the Processor template. Each actor in `SDFG` and each instantiation of `Processor` template is given an unique id and passed as parameters to the templates. Whole system is comprised

**Listing 1: System declarations**

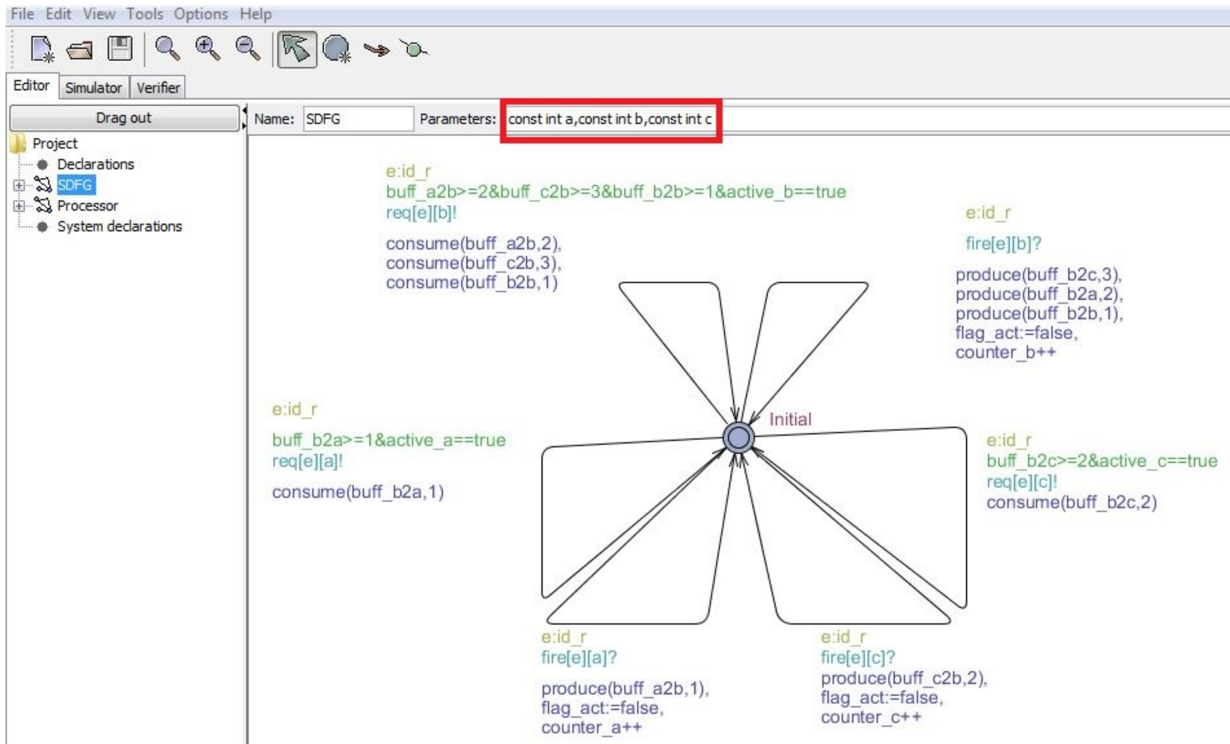
```
// Actor ids
const int a=0;
const int b=1;
const int c=2;

// Processor ids
const int p0=0;
const int p1=1;
const int p2=2;
const int p3=3;

// SDF Graph template instantiation
SDF_Graph = SDFG(a,b,c);

// Processor template instantiation
Processor0 = Processor(p0,a,b,c);
Processor1 = Processor(p1,a,b,c);
Processor2 = Processor(p2,a,b,c);
Processor3 = Processor(p3,a,b,c);

// Processes to be composed into a system.
system SDF_Graph, Processor0,Processor1, Processor2, Processor3;
```

of one instance of `SDFG` called `SDFG_Graph` and four instances of `Processor` called `Processor0, Processor1, Processor2` and `Processor3` as it is declared in Listing 1.

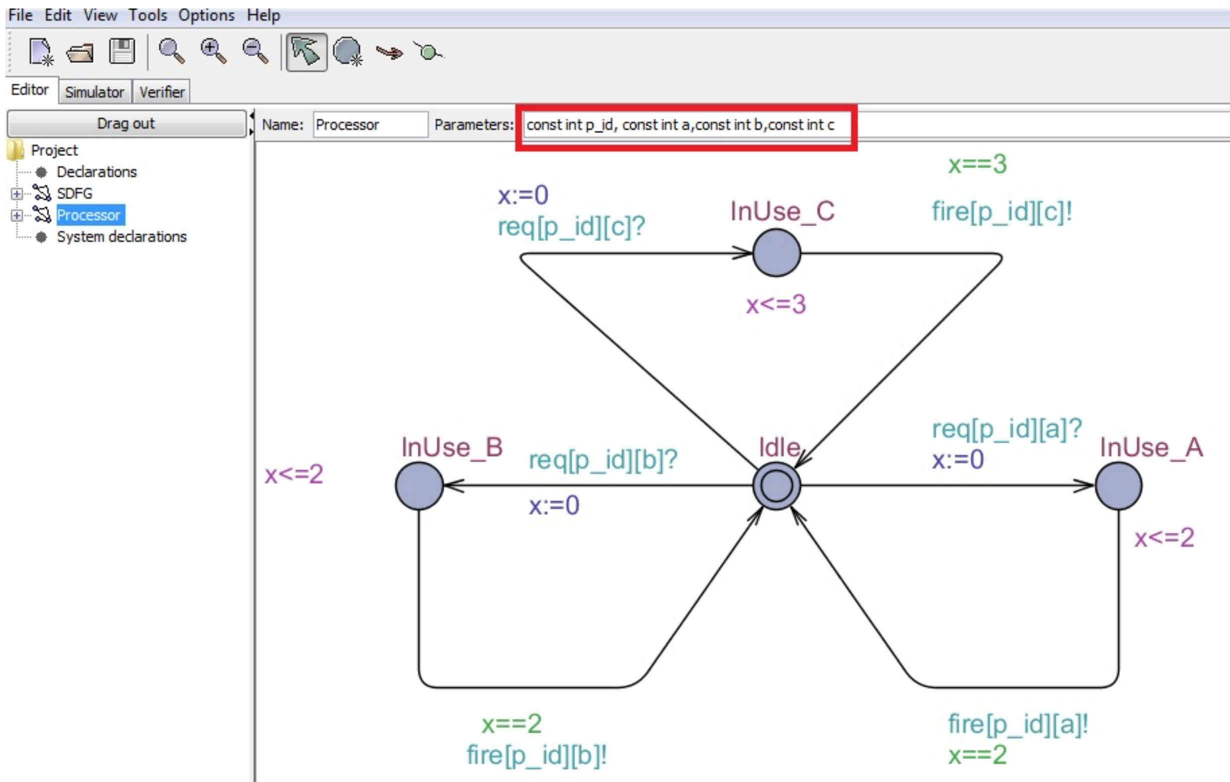
Figure 9 explains the models of `SDFG` and `Processor` in the editor of UPPAAL and Listing 2 describes all the global declarations used in these templates. There are two edges for each actor and a single location *Initial*. The parameters consist of ids of each actor. Label `e:id_r` selects the processor ids from user-defined type `id_r` declared in Listing 2 by which SDF graph template communicates with Processor template. For each edge in the SDF graph, there is an integer variable in UPPAAL model where initial value of the variable is equal to the initial number of tokens in the edge. For example, in Listing 2, initial tokens in the edge from actor $c$ to actor $b$ is defined by `int buff_c2b=6;`. The constant variable `N` and `M` denotes total number of the processors required and the actors respectively. Channels `req[N][M]` and `fire[N][M]` are used to synchronise both templates. Functions `produce` (`consume`) respectively produces (consumes) tokens equal to production (consumption) rate of the particular edge. Integer variables `counter_a, counter_b` and `counter_c` counts the number of times actor $a$,$b$ and $c$ fires respectively. Boolean variable `flag_act` has an initial value equal to true and its value changes to false as soon as any actor completes its firing. This variable is needed to calculate repetition vector. In Listing 2, `clock global` observes the overall time progress of any trace. The clock variable `x` of the processor is declared as a local variable (not shown here).

*Idle* in the Processor model in Figure 9 is an initial location and *InUse_A, InUse_B* and *InUse_C* are the dedicated locations for each actor. In this model, the processor ids are represented by `p_id` and are passed as parameters.

Figure 10 shows the simulator tab with a SDF graph and one processor automaton. Synchronisation messages between SDF graph and all four processors are shown on message sequence chart.




(a) View of SDF graph template in the UPPAAL editor




(b) View of Processor template in the UPPAAL editor

Figure 9: UPPAAL editor showing SDF graph and Processor

Listing 2: Global declarations

```
//Global Clocks
clock global;

const int N = 4;        //Number of Processors
const int M = 3 ;   //Number of Actors

//Task and Processors IDs
typedef int[0,N-1] id_r;

//Channels
chan fire[N][M], req[N][M];

//Buffer Sizes
int buff_a2b,buff_b2c=0;
int buff_b2a=2;
int buff_c2b=6;
int buff_b2b=1;

//Flag to check if SDF Graph has started executing
bool flag_act=true;

//Counter for each actor
int counter_a, counter_b, counter_c=0;

void produce(int &channel_tokens, int tokens)
{
     channel_tokens+=tokens;
}
void consume(int &channel_tokens, int tokens)
{
        channel_tokens-=tokens;
}
```

## 6.2 Throughput Calculation

Uppaal has an option of generating trace with smallest time delay called `Fastest Trace`. Exploiting this option, we can determine repetition vector and throughput. If we have Uppaal models of SDF graph and processors and if we ask Uppaal to give us fastest trace to $n^{th}$-multiple of repetition vector, Uppaal makes sure an iteration is completed in a least possible time. As a result, Uppaal returns a trace where at one point, SDF graph leaves the transient phase and enters the periodic phase and then returns back to the initial token distribution. By observing the trace, we can determine the maximal throughput. By following the same method, we can find out the best trade-off between the throughput and our desired number of processors. Value of n must be high enough to allow sufficient iterations to a SDF graph to find periodic phase.

Repetition vector and throughput are determined by using following queries.

**Repetition Vector:** E<> (Initial Token Distribution)

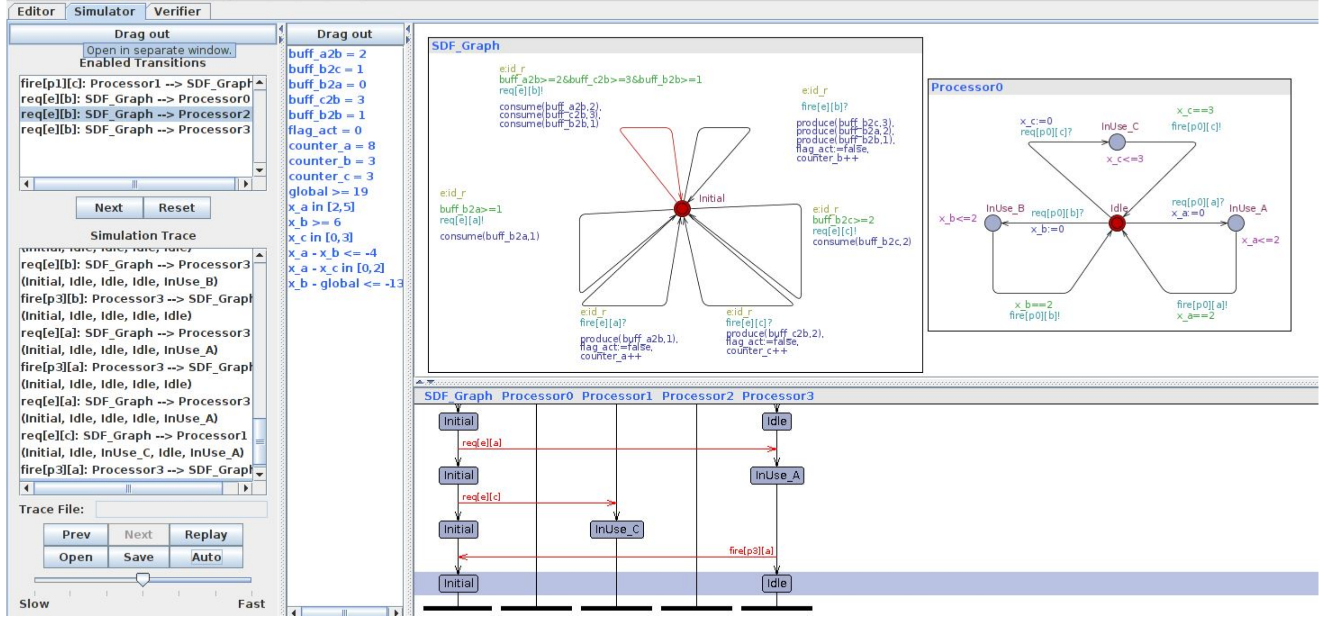



Figure 10: View of a simulation of the SDF graph-Processor model showing SDF graph and one processor

**Throughput:** E<> (Repetition Vector)

We can detect the presence or absence of deadlock in a SDF graph by following query. Due to limitations of model checkers, we have to omit all counters before checking for deadlock. All results of deadlock detection in remaining paper are calculated without any counters in the model.

**Deadlock:** A[] not deadlock

## 6.3 Results

As we know initial token distribution of the SDF graph in Figure 1, selecting `Fastest` trace and verifying the following query in Uppaal generates a trace by which we determine repetition vector.

E<> (buff_a2b==0&buff_b2c==0&buff_b2a==2&buff_c2b==6&buff_b2b==1&flag_act==false)

As a result of this query, a trace is generated and by examining the value of variables `counter_a`, `counter_b` and `counter_c` shown in Figure 11, we can determine the value of repetition vector.

As explained earlier, we can find out throughput using fifth multiple of repetition vector by verifying following query. We can analyse the generated trace to determine periodic phase and hence throughput.

E<> (counter_a==20&counter_b==10&counter_c==15)

We could determine exact number of processors required for self-timed execution which is 4 in case of our running example using SDF$^3$ tool suite. Using results presented earlier, if we reduce number of processors by 1 and model SDF graph shown in the Figure 1 with three processors in Uppaal, we get schedule portrayed in Figure 12. We can observe that even we have reduced the

Variables
buff_a2b = 0
buff_b2c = 0
buff_b2a = 2
buff_c2b = 6
buff_b2b = 1
flag_act = 0
counter_a = 4
counter_b = 2
counter_c = 3
global >= 11
x_a >= 7
x_b >= 5
x_c >= 3
x_a - global <= -4
x_a - x_b in [2,5]
x_c - x_b <= -2

Figure 11: Variables showing repetition vector

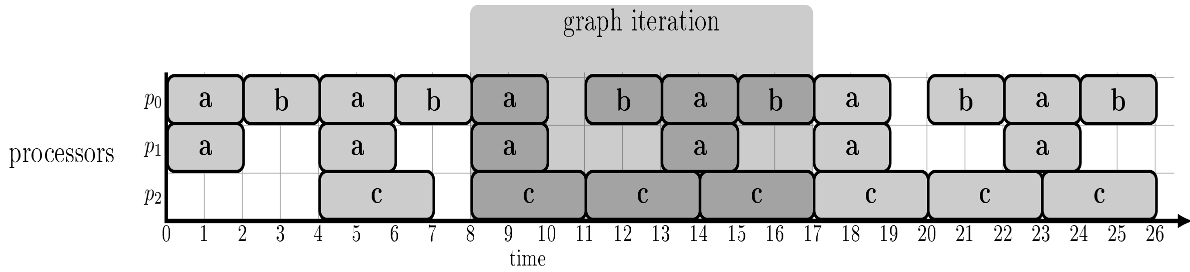



Figure 12: Scheduling using three processors

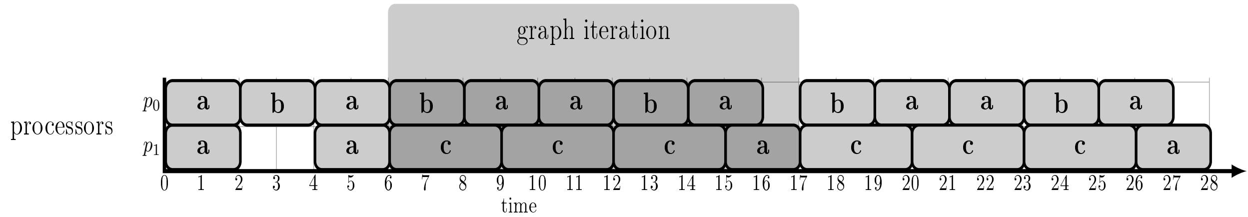



Figure 13: Scheduling using two processors

number of processors from four to three, throughput still is $\frac{1}{9}$ which clearly shows that we do not always need self-timed execution to realise maximum throughput. In the same fashion, Figure 13 shows schedule using two processors and the throughput in this case is $\frac{1}{11}$.

Table 1 records results for peak memory consumption and computation time. These figures are determined using an utility called *memtime*. The experiments were run on a dual-core 2.8 GHz machine with 4GB RAM. First column displays the number of processors, second column represents the value of throughput with respect to different number of processors. Columns 3-8 depicts memory consumption and computation time required by UPPAAL in generating the trace for determining

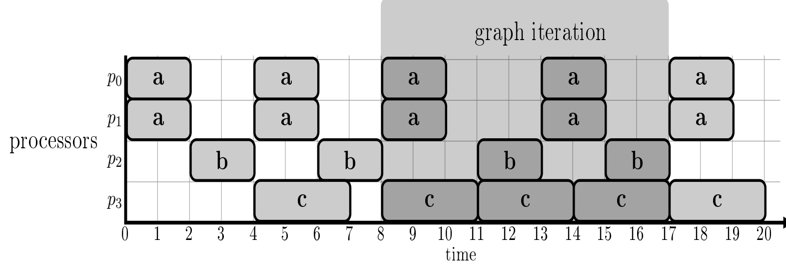



Figure 14: Scheduling in a heterogeneous system

repetition vector, throughput and deadlock freedom respectively. Last column represents time taken by SDF$^3$ tool suite for calculating throughput against 4 processors (self-timed execution). It also explains that we cannot calculate throughput of a SDF graph on less number of processors using SDF$^3$.

We have also seen that our approach generates an optimal schedule in a simple manner on a given number of processors automatically, once target state is specified in a query. We could also check efficiently if a certain SDF graph deadlocks if we reduce number of processors than required for a self-timed execution.

Table 1: Experimental Results for SDF graph in Figure 1

| **Number of Processors** | **Throughput** | **Repetition Vector** | | **Throughput** | | **Deadlock Freedom** | | **SDF3** |
|---|---|---|---|---|---|---|---|---|
| | | Memory(KB) | Time(s) | Memory(KB) | Time(s) | Memory(KB) | Time(s) | Time(ms) |
| 4 (self-timed) | 1/9 | 2008 | 0.1 | 38148 | 0.2 | 2008 | 0.1 | 0 |
| 3 | 1/9 | 2008 | 0.1 | 38012 | 0.28 | 2008 | 0.1 | - |
| 2 | 1/11 | 2008 | 0.1 | 37880 | 0.29 | 2008 | 0.1 | - |
| 1 | 1/21 | 2008 | 0.1 | 2008 | 0.1 | 2008 | 0.1 | - |

## 6.4 Scheduling in a Heterogeneous System

So far, we have assumed a homogeneous system only where an actor can be mapped on any processor as all processors are identical. A homogeneous system naturally gives more freedom to decide which actor to assign to a particular processor. On the contrary, this freedom is limited in a heterogeneous system by which processors could be utilised to execute a particular actor.

In Uppaal, we can utilise the same models explained earlier in a heterogeneous system. Let us consider a SDF graph of Figure 1 in a heterogeneous system in which actor $a$ can be mapped only on the processors p0 and p1, actor $b$ can be executed only on the processor p2 and the processor p3 is assigned to execute actor $c$ only. We change the value of variable `M` to four in Listing2 and introduce a dummy actor in "System declarations" as mentioned in Listing3. We can see in Listing3 that the dummy actor is passed as a parameter in place of those actors which are not to be bound to a particular processor The schedule of this heterogeneous system is displayed in Figure 14 and throughput is $\frac{1}{9}$.

Table 2 shows throughput, peak memory consumption and computation time for a heterogeneous system. We cannot compute throughput of an unbounded SDF graph on a heterogeneous

Listing 3: System declarations

```
// Actor ids
const int a=0;
const int b=1;
const int c=2;
const int dummy=3;

// Processor ids
const int p0=0;
const int p1=1;
const int p2=2;
const int p3=3;

// SDF Graph template instantiation
SDF_Graph = SDFG(a,b,c);

// Processor template instantiation
Processor0 = Processor(p0,a,dummy,dummy);
Processor1 = Processor(p1,a,dummy,dummy);
Processor2 = Processor(p2,dummy,b,dummy);
Processor3 = Processor(p3,dummy,dummy,c);

// Processes to be composed into a system.
system SDF_Graph, Processor0,Processor1, Processor2, Processor3;
```

system using SDF[3].

Table 2: Experimental Results for SDF graph in Figure 1 on a heterogeneous system

| **Number of Processors** | **Throughput** | **Repetition Vector** | | **Throughput** | | **Deadlock Freedom** | | **SDF3** |
|---|---|---|---|---|---|---|---|---|
| | | Memory(KB) | Time(s) | Memory(KB) | Time(s) | Memory(KB) | Time(s) | Time(ms) |
| 4 | 1/9 | 2008 | 0.1 | 2008 | 0.1 | 2008 | 0.1 | - |

## 6.5 Other Case Studies

This subsection presents results of the experiments on different case studies. We have used a bipartite graph with buffer capacities [9] in Figure 15, a MPEG-4 decoder [22] capable of processing 5 macro blocks in Figure 16, a MP3 decoder [23] in Figure 17, two example SDF graphs shown in Figure 18 and Figure 19 and an audio echo canceller [12] in Figure 20. Table 3 records repetition vector of each SDF graph and Table 4 displays the results of the experiments of finding out repetition vector, throughput and deadlock freedom and comparison with SDF[3]. We can observe in Table 4 that UPPAAL consumes less memory and time for less number of processors. It is also possible to determine trade-off between the number of processors and throughput.

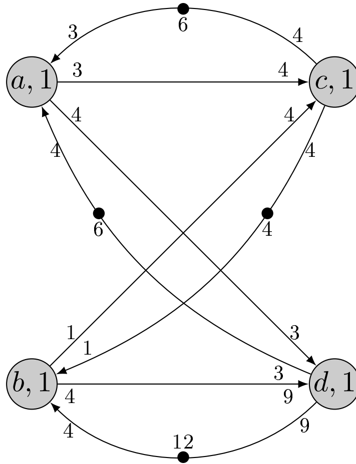



Figure 15: Bipartite Graph [9]

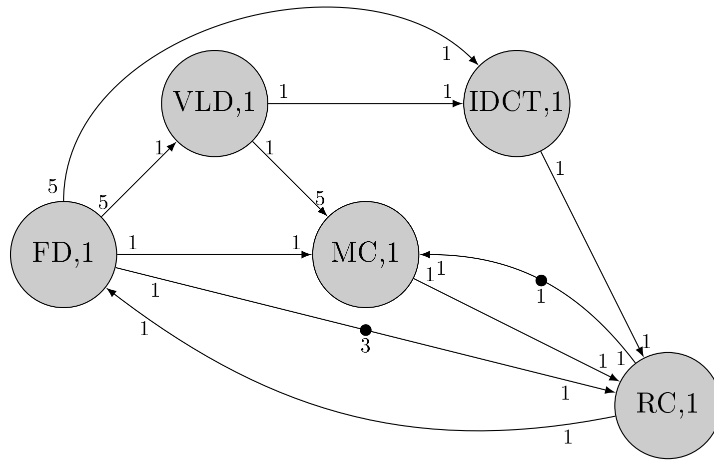



Figure 16: MPEG-4 Decoder [22]

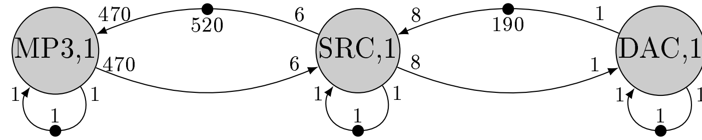



Figure 17: MP3 Decoder [23]

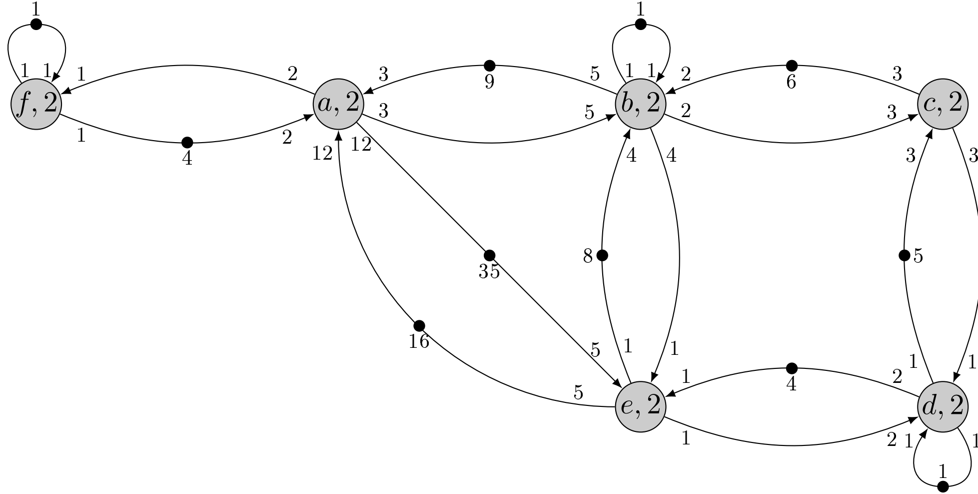



Figure 18: Example SDF Graph

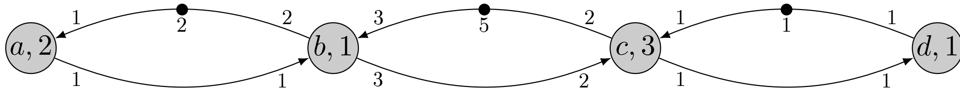



Figure 19: Example SDF Graph [11]

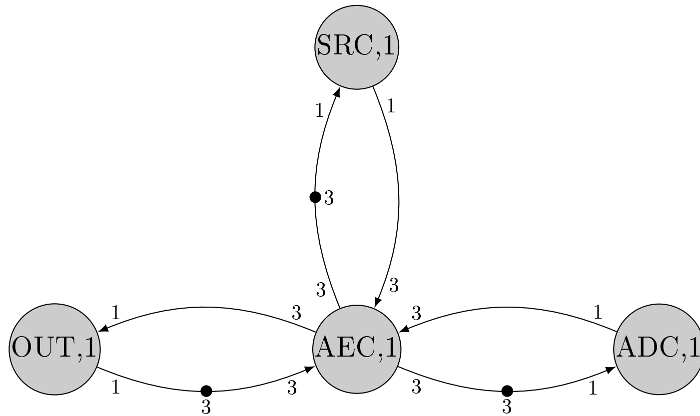



Figure 20: Audio Echo Canceller [12]

# 7 Conclusions and future work

Despite of remarkable progress in the modelling and analysis of SDF graphs, yet compact methods for the efficient scheduling of SDF graphs are needed with a best trade-off between the maximal

Table 3: Repetition Vectors

| Models | Repetition Vector |
|---|---|
| Bipartite graph in Figure 15 | [a b c d] = [12 36 9 16] |
| MPEG-4 Decoder in Figure 16 | [FD VLD IDCT RC MC] = [1 5 5 1 1] |
| MP3 Decoder in Figure 17 | [MP3 SRC DAC] = [3 235 1880] |
| Example SDF graph in Figure 18 | [a b c d e f] = [5 3 2 6 12 10] |
| Example SDF graph in Figure 19 | [a b c d] = [2 2 3 3] |
| Audio Echo Canceller in Figure 20 | [OUT SRC AEC ADC] = [3 3 1 3] |

Table 4: Experimental Results

| Number of Processors | Throughput | Repetition Vector | | Throughput | | Deadlock Freedom | | SDF3 |
|---|---|---|---|---|---|---|---|---|
| | | Memory(KB) | Time(s) | Memory(KB) | Time(s) | Memory(KB) | Time(s) | Time(ms) |
| Bipartite graph in Figure 15 | | | | | | | | |
| 4 (self-timed) | 1/42 | 38168 | 0.21 | 39352 | 0.41 | 38024 | 0.21 | 0 |
| 3 | 1/44 | 38156 | 0.2 | 38284 | 0.31 | 38008 | 0.2 | - |
| 2 | 1/51 | 2008 | 0.1 | 38032 | 0.21 | 2008 | 0.1 | - |
| 1 | 1/73 | 2008 | 0.1 | 38276 | 0.21 | 2008 | 0.1 | - |
| MPEG-4 Decoder in Figure 16 | | | | | | | | |
| 6 (self-timed) | 1/4 | 41584 | 2.14 | 55680 | 12.52 | 41576 | 3.5 | 0 |
| 5 | 1/5 | 39272 | 1.02 | 44400 | 4.75 | 39320 | 1.11 | - |
| 4 | 1/5 | 38288 | 0.3 | 40128 | 1.07 | 38268 | 0.41 | - |
| 3 | 1/6 | 2008 | 0.11 | 38300 | 0.3 | 38008 | 0.2 | - |
| 2 | 1/8 | 2008 | 0.1 | 2008 | 0.1 | 2008 | 0.11 | - |
| 1 | 1/13 | 2008 | 0.1 | 2008 | 0.1 | 2008 | 0.1 | - |
| MP3 Decoder in Figure 17 | | | | | | | | |
| 2 (self-timed) | 1/1880 | 68660 | 4.24 | 227884 | 51.80 | 67056 | 8.93 | 36.002 |
| 1 | 1/2118 | 47268 | 1 | 109192 | 5.73 | 47248 | 2.1 | - |
| Example SDF graph in Figure 18 | | | | | | | | |
| 5 (self-timed) | 1/24 | 68936 | 24.8 | 200784 | 166.57 | 71932 | 36.2 | 0 |
| 4 | 1/24 | 47936 | 5.67 | 88772 | 28.93 | 48600 | 9.66 | - |
| 3 | 1/28 | 40316 | 1.11 | 50588 | 5.15 | 40500 | 1.92 | - |
| 2 | 1/38 | 38160 | 0.2 | 40408 | 0.71 | 38284 | 0.3 | - |
| 1 | 1/76 | 2008 | 0.1 | 38700 | 0.31 | 2008 | 0.1 | - |
| Example SDF graph in Figure 19 | | | | | | | | |
| 2 (self-timed) | 1/12 | 2008 | 0.1 | 2008 | 0.1 | 2008 | 0.1 | 0 |
| 1 | 1/18 | 2008 | 0.1 | 2008 | 0.1 | 2008 | 0.1 | - |
| Audio Echo Canceller in Figure 20 | | | | | | | | |
| 6 (self-timed) | 1/2 | 38568 | 0.42 | 50176 | 4.48 | 4148 | 1.7 | 0 |
| 5 | 1/3 | 38148 | 0.21 | 42616 | 1.63 | 39176 | 0.7 | - |
| 4 | 1/3 | 2008 | 0.1 | 39220 | 0.52 | 38264 | 0.3 | - |
| 3 | 1/3 | 2008 | 0.1 | 37892 | 0.2 | 2008 | 0.1 | - |
| 2 | 1/4 | 2008 | 0.1 | 378884 | 0.2 | 2008 | 0.1 | - |
| 1 | 1/7 | 2008 | 0.1 | 2008 | 0.1 | 2008 | 0.1 | - |

throughput and number of processors. By translating SDF graphs to TA and implementation in UPPAAL, we have combined the flexibility of automata with the efficiency of SDF graphs to find

best schedules.

Moreover, with the help of contemporary model checkers like UPPAAL, benefits over the range of analysable properties like absence of deadlocks and unboundedness, safety, liveness and reachability can also be enjoyed. We encountered some limitations using UPPAAL in this context like,

- State-space explosion problem for the bigger models.
- Inability to model-check using counters and getting an error message of out-of-range assignment.
- Inability of expressing more complex statements using Leads to property such as nesting of path quantifiers.

To tackle these problems, we plan to apply multi-core reachability using LTSmin [14]. Future work also includes energy optimal reachability analysis with the help of UPPAAL CORA [1] and a possibility to extend SDF models with the features like stochastics. Similarly, we also plan to translate recent extension of SDF i.e. Scenario Aware Dataflow to TA, enrich it with minimum-cost reachability and mappings to Markov automata. This will lead us to achieve self-supporting computation in the target systems where energy generation, energy storage, and energy consumption is kept in balance over the lifetime of a system.

# References


[1] UPPAAL CORA. `http://people.cs.aau.dk/~adavid/cora/`.

[2] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, ICALP '90, pages 322–335, London, UK, UK, 1990. Springer-Verlag.

[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 M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[5] N. Bertrand, A. Stainer, T. Jéron, and M. Krichen. A game approach to determinize timed automata. In *Proceedings of the 14th international conference on Foundations of software science and computational structures*, FOSSACS'11/ETAPS'11, pages 245–259, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:889–899, 1997.

[7] 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 Software Engineering and Advanced Applications (SEAA) , Cesme, Izmir, Turkey*, pages 29–38. IEEE Computer Society, 2012.

[8] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS*, pages 372–387, 2011.

[9] M. Geilen, T. Basten, and E. Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *in Proceedings of the Design Automation Conference*, pages 819–824. ACM, 2005.

[10] A. Ghamarian, M. Geilen, T. Basten, B. Theelen, M. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *In Formal Methods in Computer Aided Desgin, FMCAD 06, Proceedings. IEEE, 2006*, pages 68–75, 2006.

[11] 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 *In ACSD 06, Proc. (2006), IEEE*, pages 25–34. IEEE, 2006.

[12] 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 *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 185–194, New York, NY, USA, 2012. ACM.

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

[14] A. Laarman, J. van de Pol, and M. Weber. Multi-core ltsmin: Marrying modularity and scalability. In *NASA Formal Methods*, pages 506–511, 2011.

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

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

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

[18] R. Reiter. Scheduling parallel computations. *J. ACM*, 15(4):590–599, 1968.

[19] S. Stuijk. Predictable mapping of streaming applications on multiprocessors. In *Phd thesis*, 2007.

[20] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 777–782, New York, NY, USA, 2007. ACM.

[21] S. Stuijk, M. Geilen, and T. Basten. SDF$^3$: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006.

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

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

[24] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs. In *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, pages 96–105, 2009.

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