Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications

Adnan Bouakaz, Jean-Pierre Talpin, Jan Vitek

Affine Data-Flow Graphs for the Synthesis of Hard Real-Time Applications

Adnan Bouakaz
University of Rennes 1 / IRISA
Rennes, France
adnan.bouakaz@irisa.fr

Jean-Pierre Talpin
INRIA / IRISA
Rennes, France
jean-pierre.talpin@inria.fr

Jan Vitek
Purdue University
West Lafayette, Indiana, USA
jv@cs.purdue.edu

***Abstract*—Data-flow models ease the task of constructing feasible schedules of computations and communications of high-assurance embedded applications. One key and open issue is how to schedule data-flow graphs so as to minimize the buffering of data and reduce end-to-end latency. Most of the proposed techniques in that respect are based on either static or data-driven scheduling. This paper looks at the problem in a different way by considering priority-driven preemptive scheduling theory of periodic tasks to execute a data-flow program.**

Our approach to the problem can be detailed as follows. (1) We propose a model of computation in which the activation clocks of actors are related by affine functions. The affine relations describe the symbolic scheduling constraints of the data-flow graph. (2) Based on this framework, we present an algorithm that computes affine schedules in a way that minimizes buffering requirements and, in addition, guarantees the absence of overflow and underflow exceptions over communication channels. (3) Depending on the chosen scheduling policy (earliest-deadline first or rate-monotonic), we concretize the symbolic schedule by defining the period and the phase of each actor. This concretization guarantees schedulability and maximizes the processor utilization factor.

***Keywords*-Data-flow graphs, Buffer minimization, Affine relation, Priority-driven scheduling, Linear programming.**

I. Introduction

Embedded systems are playing a crucial role in our life, they are used in chemical and nuclear plants, aircraft flight control systems, military systems, etc. The key properties of such systems are functional determinism and schedule feasibility. Functional determinism means that, for a given set of inputs, the system will always produce the same set of outputs. Schedule feasibility means that the system will meet its deadlines even in the worst case scenario. Ensuring those properties is difficult in case shared-memory and traditional lock-based mutual exclusion protocols are used for concurrency control. We propose to investigate a data-flow concurrency model in order to exclude potential for concurrency errors and race conditions. Furthermore, the concurrency model we propose aims at simplifying the task of synthesizing feasible schedules.

In that application and for this aim, data-flow graphs offer simple modeling concepts to ease the engineering of software and hardware architectures by waiving the burden of explicitly specifying schedules for computations and communications thanks to automated synthesis techniques that can be developed in that framework [1]. A data-flow graph models a program by of a set of actors communicating through one-to-one FIFO channels. Hence, concurrency can be implemented without explicit synchronization mechanisms and data races can be avoided at compile-time. Previous experiments with data-flow modeling for real-time Java with StreamFlex [2] and FlexoTask [3] demonstrated its potential to assist software engineering of safety critical applications with automated code generation techniques. Case studies additionally helped to learn the need for a precise semantic and an analytic framework to precisely determine the size of communication channels between tasks and to decide schedule feasibility.

In its seminal paper [4], Kahn provides a denotational semantics of data-flow programs and shows under which conditions a network of stream processes (so-called KPN) is *deterministic* regardless of the way communications and computations are scheduled in the network (i.e. in a latency-insensitive manner). Any implementation strategy of KPNs must obey a Kahn semantics [5] regardless of the chosen static, data-driven, or demand-driven scheduling policy. In a data-driven scheduling policy, an actor executes whenever enough tokens are available on its input ports. In a demand-driven scheduling policy, an actor executes when its outputs are needed by another actor. Those scheduling policies tend to complicate the schedulability analysis.

We consider a model of Affine Data-Flow (ADF) graphs in which each actor consists of a set of input and output ports, and ports are connected to each other via one-to-one FIFO channels. An actor is associated with an activation clock and executes at each clock tick. Any pair of clocks in the network can be related by an affine function. This model makes schedulability analysis straightforward by using real-time scheduling theory. In addition, by proving that our model of computation conforms to a Kahn semantics, we ensure functional determinism.

Related Work

Special cases of KPNs that can be executed with bounded channels have been extensively studied, especially for cyclo-static data-flow (CSDF) graphs [6] and synchronous data-flow (SDF) graphs [7], two well-known frameworks for

embedded system design. Actors of a SDF graph consume and produce a fixed number of tokens on a given port each time they are executed. In the CSDF model, this number of tokens changes from one time to another in a cyclic manner. A comparison between SDF and CSDF can be found in [8]. As a generalization of those models, we study a more expressive class of data-flow graphs, where the number of processed tokens is described by an ultimately periodic sequence.

Minimization of buffering requirements in (C)SDF graphs under throughput constraints has been addressed in previous work [9]–[11]. In [10], authors present a polynomial heuristic that computes static-periodic schedules of actors under throughput constraints in order to minimize buffer capacities. However, the algorithm does not construct a global and unique schedule of actors since it results on as much schedules of an actor as its adjacent buffers. This problem was tackled in [11].

In a way akin to related works, one of our goals is to minimize the size of communication buffers, but our approach explores a new path which differs in three respects. (1) The computed buffer capacities are machine-independent: they are based on symbolic affine schedules of the data-flow graph. (2) We target classical priority-driven scheduling policies such as Rate-Monotonic (RM) and Earliest-Deadline First (EDF) scheduling policies: we guarantee schedulability conditions under each of these policies by synthesizing the appropriate timing characteristics of actors (i.e. periods and phases) in a way that maximizes processor utilization. (3) We provide more freedom when writing the implementation code of actors: we do not make any assumption on which order an actor produces and consumes tokens, i.e. an actor can consume and produce tokens whenever it wants. This freedom comes at the price of a more conservative analysis.

In [12], the EDF scheduling theory is applied to a subset of the Processing Graph model (PGM). In that work, the execution rates of each node is defined as a function of the input rates. Their data-flow model was constrained: data-flows must form acyclic connected graphs, a constant amount of data can be processed at each execution step, tokens are written/read atomically, and consumption takes place after production. In addition, the semantic model is data-driven, i.e. an actor is executed whenever the number of accumulated tokens on a channel exceeds a given threshold. Therefore, an actor cannot be modeled by a periodic task. Hence, instead of using a periodic task model, RBE task model is used [13], where scheduling analysis is done according to a processor demand approach. However, this results in a more costly pseudo-polynomial complexity than with the processor utilization approach we adopt.

Finally, the work presented in [14] is closely related to our approach in that data-flow graphs are scheduled as periodic task systems. However, the proposed method applies to acyclic connected CSDF graphs, and the computed buffer capacities are machine-dependent. Also, the graphs are scheduled on a multi-processor systems, but our approach can be easily extended to that case by only changing the bound on the processors utilization factor in the schedulability test.

Safety-Critical Java

This paper shows how to automatically generate a Safety-Critical Java (SCJ) application from a data-flow specification. In this context, the choice of SCJ [15] is justified by the built-in priority-driven preemptive scheduler of its virtual machine. The SCJ specification is a domain-specific API of Java which aims at the development of qualified safety-critical applications. To better meet domain-specific safety requirements, the SCJ specification defines three levels of compliance, each with a different model of concurrency, each aiming at applications of specific criticality. In this paper, we focus on Level 1 SCJ, where the scheduler obeys a priority-driven preemptive policy. A Level 1 SCJ program is organized as a sequence of missions. A mission starts in an *initialization* phase during which a set of schedulable objects (i.e. periodic and aperiodic event handlers) are created. These objects are released during the mission *execution* phase, and terminated during the *cleanup* mission. A schedulable object is a bounded asynchronous event handler defined by a computation logic and some scheduling constraints. The computation logic is implemented in the `handleAsyncEvent()` method which is executed every time the schedulable object is released. A simple example of SCJ applications is the `miniCDj` benchmark [16]. In this paper, we more specifically focus on uniprocessor systems with periodic event handlers (PEHs) and propose to map each actor in the specification to a PEH in the implementation.

Plan

The paper is organized as follows. The affine data-flow model is presented in Section II. Section III outlines the three important analyses of ADF graphs, namely consistency, overflow, and underflow analyses. Section IV describes the synthesis of affine relations and the timing synthesis algorithm. By applying those algorithms on the MP3 playback case study from [9]–[11], we investigate their accuracy in Section V. We then present the SCJ code generated for that application before concluding in Section VI.

II. Affine Data-flow graphs

An affine data-flow graph is a disconnected directed graph of actors. An actor consists of a set of input ports, a set of output ports, and a firing function. Output ports are connected to input ports via one-to-one FIFO channels. Each actor p in the graph is associated with an activation clock $\hat{p}$ (an infinite ordered set of ticks). Actor p fires at every tick $\hat{p}_t$, and the execution of its firing function must terminate

before the subsequent tick $\hat{p}_{t+1}$. So, scheduling of actors is neither data-driven nor demand-driven, but it will be time-triggered.

The activation clocks are manipulated as abstract clocks, i.e. the actual duration between two successive ticks does not matter and it is not assumed to be constant. Later in this section, we will specify some relations between activation clocks.

Self loops are authorized in the graphs because they fit naturally in a Kahn semantics [4], [17]. They allow modeling of local variables, however they enforce a precedence relationship between successive firings of an actor, i.e. an actor fires only after termination of the previous firing. This will ensure proper state updates. In the subsequent, we will omit self loops from the graphs since we have already imposed the precedence relationship between successive firings.

An actor is usually constrained, for example in [9]–[11], to read all the required data before executing the firing function and to write the results only after the execution finishes. We get rid of this constraint in the ADF model, i.e. an actor may consume and produce tokens whenever it wants. The rational behind this choice is to give the designer more freedom when writing the Java implementation code of the firing functions. This freedom, however, comes at the price of a conservative analysis.

The number of tokens consumed or produced during firings are indicated by some functions called *amplitude functions*.

Definition 1 (Amplitude function)**.** *An amplitude function, g_x associated with port x, is a bounded integer function $g_x : \mathbb{N} \longrightarrow \mathbb{N}$ such that $\forall j \in \mathbb{N}, \alpha_x \leq g_x(j) \leq \beta_x$* [1].

During the j^{th} firing, an actor consumes $g_y(j)$ tokens from every input port y, and produces $g_x(j)$ tokens on every output port x. Amplitude functions must be bounded, otherwise the network of actors cannot be a KPN [17]. An amplitude function is *static* in the sense that the number of consumed or produced tokens is data-independent (it has a unique argument: its firing count j).

Two activation clocks can be related by a firing relation which expresses the rate of activation of an actor relative to another. Firing relations describe an abstract scheduling of the data-flow graph and can be formally defined as follows.

Definition 2 (firing relation)**.** *A firing relation between two actors p and q is defined by two **monotonic** functions: $\mathcal{R}_{p,q}, \mathcal{R}_{q,p} : \mathbb{N} \longrightarrow \mathbb{N}$ such that $\forall j \in \mathbb{N}$, $\mathcal{R}_{p,q}(j)$ (resp. $\mathcal{R}_{q,p}(j)$) is the number of firings of q (resp. p) that happened **before** the j^{th} firing of p (resp. q).*

The two monotonic functions must be coherent with each other, i.e.

$$\forall j \in \mathbb{N}, \mathcal{R}_{p,q}(j) = j' > 0 \Rightarrow \mathcal{R}_{q,p}(j' - 1) \leq j$$

$$\forall j' \in \mathbb{N}, \mathcal{R}_{q,p}(j') = j > 0 \Rightarrow \mathcal{R}_{p,q}(j - 1) \leq j'$$

As said before, the durations between ticks do not matter, the most important is the *relative positioning of ticks*. In Figure 1, $\mathcal{R}_{p,q_1}(0) = 1$ which means that actor q_1 fires one time before the 0^{th} firing of actor p (i.e. before $\hat{p}_0$). Note that $\mathcal{R}_{p,q_1}$ is equivalent to $\mathcal{R}_{p,q_2}$ but $\mathcal{R}_{q_1,p}(6) \neq \mathcal{R}_{q_2,p}(6)$. Indeed, $\hat{p}_2$ is *synchronous* with the 5^{th} firing of q_1 but not with that of q_2. Hence, we need two functions to describe a firing relation.

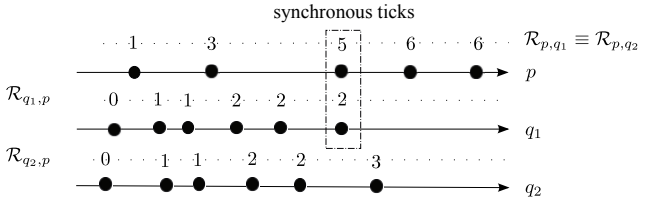

Figure 1. Example of firing relations.

A firing relation must exist between every pair of adjacent actors in the data-flow graph. However, one may impose further firing relations between unconnected actors. Our aim is to synthesize all the necessary firing relations in accordance with user-imposed functional and temporal requirements.

A. Correctness of the implementation strategy

In this subsection and in order to guarantee functional determinism, we investigate whether our MoC conforms to a Kahn semantics or not. The implementation strategy is correct if it satisfies the following three correctness criteria [5]: boundedness, completeness, and soundness.

Boundedness: The implementation strategy is *bounded* if it produces a bounded executive *whenever this latter exists*. A bounded executive is such that the number of unconsumed tokens in every internal channel and in every execution step cannot exceed a constant bound.

Let $c = (x, y)$ be a channel that connects the output port x of an actor p to the input port y of an actor q. It has a size equal to $h(c)$ and may be initialized by $\bar{c}$ tokens. Boundedness implies that if the producer fires j times and the consumer fires j' times, then the number of accumulated tokens on channel c is less or equal to $h(c)$. Certainly, j and j' must be related by a firing relation between p and q.

Let us define a new integer function G_x such that $\forall j \in \mathbb{N}, G_x(j) = \sum_{i=0}^{j} g_x(i)$. This function denotes the total amount of consumed or produced tokens on port x until the j^{th} firing of the actor. So, boundedness means that for every channel $c = (x, y)$ we have that,

$$\forall (j, j'), \bar{c} + G_x(j) - G_y(j') \leq h(c) \tag{2}$$

An overflow exception will be thrown when an actor attempts to write to a full channel. If we prove *statically* that

[1] for two constants $\alpha_x = \min g_x$ and $\beta_x = \max g_x$

Equation 2 is satisfied, then we guarantee that the execution of the data-flow graph will be free from overflow exceptions. The overflow analysis is presented in details in Section III.

Completeness: It means that the stream produced incrementally on each output converges to the stream specified by the denotational semantics. Completeness implies that no process may starve. In our MoC, a process (constructed from successive firings of an actor) cannot starve because its corresponding actor fires infinitely according to its activation clock which is an infinite set of ticks. However, the execution is not complete if a memory exception is thrown. Overflow exceptions are excluded by Equation 2, and in order to exclude underflow exceptions (i.e. when an actor attempts to read from an empty channel), we have to ensure that: for every channel $c = (x, y)$, if the consumer fires j' times and the producer fires j times, then the number of accumulated tokens on channel c cannot be negative. Again, here, j' and j are related by a firing relation. Formally,

$$\forall (j', j), \bar{c} + G_x(j) - G_y(j') \geq 0 \tag{3}$$

It is worth mentioning that Equation 3 may reject some data-flow graphs. In fact, if there is a (partial) deadlock in a graph according to the Kahn blocking read semantics, then its execution may cause an underflow exception.

Soundness: The stream produced on each output is a prefix of the stream specified by the denotational semantics. This requirement is clearly satisfied in our MoC.

B. Classes of amplitude functions

If all the amplitude functions are constant (i.e. $\forall j \in \mathbb{N}, g_x(j) = \alpha_x$ such that x is a port), then the unclocked version of the graph is a SDF graph [7].

If all the amplitude functions are periodic (i.e. $\exists \pi_x \in \mathbb{N}^+ \forall j \in \mathbb{N}, g_x(j) = g_x(j + \pi_x)$ such that x is a port), then the unclocked version of the graph is a CSDF graph [6].

In this paper, we define a more general class of amplitude functions: ultimately periodic functions (i.e. $\exists \pi_x \in \mathbb{N}^+ \ \exists j_x \in \mathbb{N} \ \forall j \geq j_x, g_x(j) = g_x(j + \pi_x)$ such that x is a port). For conciseness, we use ultimately periodic sequences (defined below) to denote those amplitude functions.

Definition 3 (Ultimately periodic sequence)**.** *Let $s \in \mathbb{N}^\omega$ be an infinite integer sequence. The sequence s is ultimately periodic if and only if it is composed of a prefix $u \in \mathbb{N}^*$ followed by a sequence $v \in \mathbb{N}^*$ repeated infinitely. When this is the case, we write $s = u(v)$.*

So, $g_x = u(v)$ means that:

$$\forall j \in \mathbb{N}, g_x(j) = \begin{cases} u[j] & \textit{if } j < |u| \\ v[(j - |u|) \mod |v|] & \textit{otherwise} \end{cases}$$

If $u \in \mathbb{N}^*$ is a finite integer sequence, then $|u|$ denotes its length and $||u||$ denotes the sum of its elements. For an amplitude function denoted by an ultimately periodic sequence $s = u(v)$, we impose that $||v|| > 0$.

In the following sections, we will conduct our analyses on data-flow graphs with ultimately periodic amplitude functions. We generally manipulate the functions G_x instead of g_x. Since g_x is a bounded integer function, we can over- and under-approximate G_x by linear bounds. If $g_x = u(v)$, then we can find $\lambda_1, \lambda_2 \in \mathbb{Q}$ such that $\forall j \in \mathbb{N}, G^l_x(j) = \frac{||v||}{|v|} j + \lambda_1, G^u_x(j) = \frac{||v||}{|v|} j + \lambda_2$, and $G^l_x(j) \leq G_x(j) \leq G^u_x(j)$.

Example 1. The input port of actor p_2 in Figure 2 is associated with an amplitude function $g = 2, 0, 1(2, 1, 0, 2)$. The linear lower bound of G is $G^l(j) = \frac{5}{4} j - \frac{1}{4}$, and the linear upper bound is $G^u(j) = \frac{5}{4} j + 2$.

It is worth mentioning that to extend the model with other classes of amplitude functions, only the linear lower and upper bounds of G_x are required.

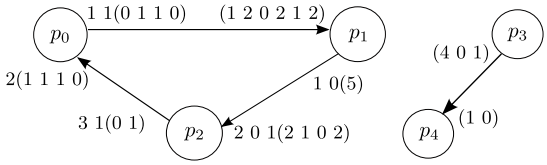

Figure 2. Example of affine data-flow graphs.

C. Affine relations

Our data-flow graphs are intended to be executed on one processor with a priority-driven preemptive scheduler. Each actor is implemented as a periodic task with a period, a phase, and a deadline equal to the period.

Let p and q be two actors. Actor p has a period of 25 *ms*, while actor q has a period of 15 *ms* and a phase of 30 *ms*. Figure 3 shows the absolute release times of p and q. So, the j^{th} release of p occurs at $25j$ *ms*, while the j'^{th} release of q occurs at $15j' + 30$ *ms*. If we ignore the actual duration between releases, we obtain a firing relation between actors p and q. The process of going from a physical time to a logical one is called *time abstraction*.

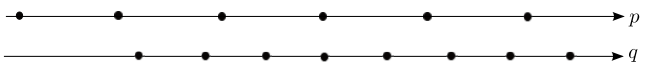

Figure 3. Releases of two periodic tasks.

In the following, we will define a special class of firing relations called *affine relations* that allows to abstract the previous periodic releases of tasks. However, affine relations are more expressive since the duration between ticks of clocks is not necessarily constant.

Definition 4 (Affine relation)**.** *As defined in [18], an affine transformation of parameters (n, φ, d) applied to the clock $\hat{p}$ produces a clock $\hat{q}$ by inserting $(n - 1)$ instants between*

any two successive instants of $\hat{p}$, and then counting on this fictional set each d^{th} instant, starting with the φ^{th} instant.

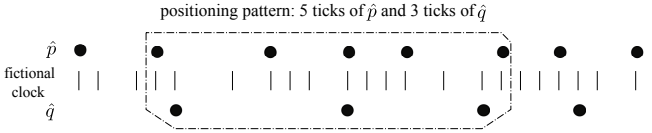

Figure 4. A $(3, 4, 5)-$affine relation.

Figure 4 shows an example of a $(3, 4, 5)-$affine relation. As one can notice, there is a positioning pattern of ticks that repeats infinitely. We say that p and q are (n, φ, d)-affine-related (or equivalently, q and p are $(d, -\varphi, n)$-affine-related), and we have that:

$$\forall j \in \mathbb{N}, \mathcal{R}_{p,q}(j) = \begin{cases} 0 & \textit{if } nj \leq \varphi \\ \lceil \frac{nj-\varphi}{d} \rceil & \textit{otherwise} \end{cases}$$

$$\forall j' \in \mathbb{N}, \mathcal{R}_{q,p}(j') = \begin{cases} 0 & \textit{if } dj' \leq -\varphi \\ \lceil \frac{dj'+\varphi}{n} \rceil & \textit{otherwise} \end{cases}$$

The sign $\lceil x \rceil$ refers to the smallest integer not less than x. Parameters n and d are strictly positive integers while φ can be negative. When defined as before, functions $\mathcal{R}_{p,q}$ and $\mathcal{R}_{q,p}$ satisfy all conditions of a firing relation.

Figure 3 can be seen as a $(25, 30, 15)-$affine relation between p and q, but also as a $(5, 6, 3)-$affine relation. So, many affine transformations can refer to the same firing relation. Thus, we will use the canonical form of affine relations presented in [18]. For an affine relation (n, φ, d) and $k = \gcd(n, d)$, there exists a canonical form *CF* defined as follows:

- $k|\varphi \Rightarrow \textit{CF}_{(n,\varphi,d)} = (\frac{n}{k}, \frac{\varphi}{k}, \frac{d}{k})$.
- $k \not| \varphi \wedge \varphi > 0 \Rightarrow \textit{CF}_{(n,\varphi,d)} = (2\frac{n}{k}, 2[\frac{\varphi}{k}] + 1, 2\frac{d}{k})$.
- $k \not| \varphi \wedge \varphi < 0 \Rightarrow \textit{CF}_{(n,\varphi,d)} = (2\frac{n}{k}, 2[\frac{\varphi}{k}] - 1, 2\frac{d}{k})$.

III. Analysis of ADF graphs

The input to our analysis is a data-flow graph, as depicted in Figure 2. The *static* analyses, which guarantee correct execution of the graph, check its consistency, and overflow and underflow freedom. All the theoretical results of this section are used in our algorithms, presented in Section IV.

A. Consistency analysis

If we synthesize each affine relation independently of the others, then the graph may be inconsistent. Indeed, assume that p, q, and r are three actors connected to each other by channels. Using the boundedness criterion (defined later in this section), we may find that $p \xrightarrow{(2,\varphi_1,3)} q \xrightarrow{(5,\varphi_2,2)} r \xrightarrow{(7,\varphi_3,5)} p$. Those three relations are inconsistent, because for three activations of p there are two activations of q, for two activations of q there are five activations of r, but for five activations of r there are seven activations of p and not three.

Proposition 1. *The graph is consistent if for every fundamental cycle $p_0 \xrightarrow{(n_0,\varphi_0,d_0)} p_1 \xrightarrow{(n_1,\varphi_1,d_1)} \cdots \rightarrow p_{m-1} \xrightarrow{(n_{m-1},\varphi_{m-1},d_{m-1})} p_0$ in the graph, we have that:*

$$\prod_{i=0}^{m-1} n_i = \prod_{i=0}^{m-1} d_i \tag{5}$$

$$\sum_{i=0}^{m-1} (\prod_{j=0}^{i-1} d_j)(\prod_{j=i+1}^{m-1} n_j)\varphi_i = 0 \tag{6}$$

proof: Let us put $\psi_i = (\prod_{j=0}^{i-1} d_j)(\prod_{j=i+1}^{m-1} n_j)$. Actors p_0 and p_1 are $(\psi_0 n_0, \psi_0 \varphi_0, \psi_0 d_0)-$affine-related. According to Definition 4, clock $\hat{p}_0$ is obtained by counting on a fictional clock $\hat{c}$ each $(\psi_0 n_0)^{th}$ instant starting with the 0^{th} instant, and $\hat{p}_1$ is obtained by counting each $(\psi_0 d_0)^{th}$ instant of $\hat{c}$ starting with the $(\psi_0 \varphi_0)^{th}$ instant. Similarly, actors p_1 and p_2 are $(\psi_1 n_1, \psi_1 \varphi_1, \psi_1 d_1)-$affine-related. So, clock $\hat{p}_1$ can be obtained by counting each $(\psi_1 n_1)^{th}$ instant of a fictional clock $\hat{c}'$. But, $\psi_1 n_1 = \psi_0 d_0$ which implies that we may used clock $\hat{c}$ instead of $\hat{c}'$. Now, clock $\hat{p}_1$ is obtained by counting each $(\psi_1 n_1)^{th}$ instant of $\hat{c}$ staring with the $(\psi_0 \varphi_0)^{th}$ instant, and clock $\hat{p}_2$ is obtained by counting each $(\psi_1 d_1)^{th}$ instant of $\hat{c}$ starting with the $(\psi_0 \varphi_0 + \psi_1 \varphi_1)^{th}$ instant. From the affine relation between p_{m-1} and p_0, we have that clock $\hat{p}_0$ is obtained by counting each $(\psi_{m-1} d_{m-1})^{th}$ instant of $\hat{c}$ starting with the $(\sum_{i=0}^{m-1} \psi_i \varphi_i)^{th}$ instant. But, we already said that clock $\hat{p}_0$ is obtained by counting each $(\psi_0 n_0)^{th}$ instant of $\hat{c}$ starting with the 0^{th} instant. So, to be consistent, it is a sufficient condition to impose Equations 5 and 6. □

It is worth mentioning again that actors, in the ADF model, consume and produce tokens whenever they want. In the absence of any knowledge on their source code, the solution is either to force some orders on reads and writes, or to perform a *conservative* analysis based on the worst-case scenarios. We opt for the second approach.

For the overflow analysis, the worst-case scenario occurs when consumption happens at the end of firings (i.e. just before the next tick), while production happens at the beginning. For the underflow analysis, the worst-case scenario is when the production happens at the end of firings, while consumption happens at the beginning.

The drawback of this conservative approach is that it increases the required buffer sizes, nevertheless it provides complete freedom when writing the implementation code of actors. Additionally, it relieves us from performing a causality analysis to detect cycles, because the tokens produced by an actor on a given firing cannot be involved in the construction of its consumed tokens at the same firing.

B. Overflow analysis

No overflow over a channel $c = (x, y)$ between the (n, φ, d)-affine-related actors p and q means that $\forall (j, j'), \bar{c}+$

$G_x(j) - G_y(j') \leq h(c)$. Only reads during firings of q that terminate before $\hat{p}_j$ are guaranteed to happen before p writes some results of its j^{th} firing. The last firing of q that terminates before $\hat{p}_j$ is $j' = \mathcal{R}_{p,q}(j) - 1$ if $\hat{q}_{\mathcal{R}_{p,q}(j)}$ is synchronous with $\hat{p}_j$, and $j' = \mathcal{R}_{p,q}(j) - 2$ otherwise.

We linearize Equation 2 to make computations more efficient, but at the cost of getting a conservative approximation. In the following, we suppose that $g_x = u_1(v_1)$ and $g_y = u_2(v_2)$.

Proposition 2. *For a given j^{th} firing of actor p, the linear lower bound of j' in the overflow analysis is $j' = \frac{n}{d}j + \frac{1-2d-\varphi}{d}$.*

proof: Since the affine relation is in canonical form, we have that $\gcd(n,d) = 1$ or $\gcd(n,d) = 2 \wedge 2 \nmid \varphi$. There are two cases:
1^{st} **case**($\hat{q}_{\mathcal{R}_{p,q}(j)}$ is synchronous with $\hat{p}_j$): This is possible only if equation $jn = kd + \varphi$ accepts many solutions. This Bézout's identity is solvable only in the case of $\gcd(n,d) = 1$, which implies that $d|(nj - \varphi)$. Therefore, the linear lower bound of j' is $\frac{n}{d}j - \frac{\varphi+d}{d}$.
2^{nd} **case**($\hat{q}_{\mathcal{R}_{p,q}(j)}$ is not synchronous with $\hat{p}_j$): The lower bound of $\mathcal{R}_{p,q}(j)$ is $\frac{n}{d}j - \frac{\varphi-1}{d}$ (since equation $nj = kd + \varphi - 1$ is always solvable). Therefore, the linear lower bound of j' is $\frac{n}{d}j - \frac{\varphi+2d-1}{d}$.

By taking the worst of the two cases, we deduce that the linear lower bound of j' is $\frac{n}{d}j - \frac{\varphi+2d-1}{d}$. □

$G_x(j)$ is approximated by the linear *upper* bound $G^u_x(j) = \frac{||v_1||}{|v1|}j + \lambda_1$; and $G_y(j')$ is approximated by the linear *lower* bound $G^l_y(j') = \frac{||v2||}{|v2|}j' + \lambda_2$ if $j' \geq 0$, and by 0 otherwise. By substituting all the linear approximations in Equation 2, we obtain the following linear constraint: $\forall j \in \mathbb{N}$,

$$\bar{c} - h(c) + \frac{||v_2||}{|v_2|d}\varphi + \xi j \leq \min\{0, \lambda_2\} - \lambda_1 + \frac{||v_2||(1-2d)}{|v_2|d} \tag{7}$$

such that $\xi = \frac{||v_1||}{|v_1|} - \frac{||v_2||n}{|v_2|d}$. Since j tends to infinity, it is a requirement for an execution free of overflows and underflows that ξ equals zero. Consequently, the boundedness criterion is:

$$\frac{n}{d} = \frac{||v_1||}{|v_1|}\frac{|v_2|}{||v_2||} \tag{8}$$

C. Underflow analysis

The underflow analysis is (roughly) a dual of the overflow analysis. No underflow over the channel c means that $\forall(j', j), \bar{c} + G_x(j) - G_y(j') \geq 0$. At its j'^{th} firing, actor q consumes only tokens produced by p on firings that finished before $\hat{q}_{j'}$. The last firing of p that terminates before $\hat{q}_{j'}$ is $j = \mathcal{R}_{q,p}(j') - 1$ if $\hat{p}_{\mathcal{R}_{q,p}(j')}$ is synchronous with $\hat{q}_{j'}$, and $j = \mathcal{R}_{q,p}(j') - 2$ otherwise.

To speed up the analysis, we proceed with a conservative linearization of Equation 3. For the j'^{th} firing of actor q, the linear lower bound of j is $\frac{d}{n}j' + \frac{1+\varphi-2n}{n}$. The proof is similar to that of proposition 2. $G_y(j')$ is approximated by the linear *upper* bound $G^u_y(j') = \frac{||v2||}{|v2|}j' + \lambda_4$; and $G_x(j)$ is approximated by the linear *lower* bound $G^l_x(j) = \frac{||v_1||}{|v1|}j + \lambda_3$ if $j \geq 0$, and by 0 otherwise.

By substituting all the linear approximations in Equation 3, we obtain the following linear constraint: $\forall j' \in \mathbb{N}$,

$$\bar{c} + \frac{||v_1||}{|v_1|n}\varphi + \frac{d}{n}\xi j' \geq \max\{0, -\lambda_3\} + \lambda_4 - \frac{||v_1||(1-2n)}{|v_1|n} \tag{9}$$

IV. Algorithms

The input to our algorithm is a data-flow graph in which each actor p is associated with a worst-case execution time *WCET*(p), and each port is associated with an ultimately periodic sequence. One may also explicitly specify additional information like channel sizes, incomplete affine relations, periods of actors, bounds on periods, etc. The algorithm proceeds in three following steps.

A. Step 1: Consistency verification

We start by performing an *abstraction* of the specified timing characteristics (user-imposed timing requirements): If periods of two actors p and q are imposed then we add an *incomplete* affine relation between them. Incomplete affine relation means that its parameter φ is undetermined. Parameters n and d of that relation are deduced from equation $n\pi_q = d\pi_p$ where π_θ is the period of actor θ.

For each channel in the graph, if an incomplete affine relation is (explicitly) specified between the producer and the consumer, then we use Equation 8 to verify the channel's boundedness; otherwise we compute n and d of the affine relation. After computing all the possible incomplete affine relations, we use Equation 5 to check the consistency of every fundamental cycle in the graph of affine relations.

B. Step 2: Synthesis of affine relations

We provide two solutions for computing the parameter φ of every incomplete affine relation in the graph in such a way we minimize the sum of channel sizes. One is exact but enumerative, the other is faster but approximate.

An enumerative solution: This exact solution is used to investigate the accuracy of the other solution. Let p and q be two $(n, \varphi, d)-$affine-related actors such that φ is undetermined and $\gcd(n,d) = 2$. Parameter φ can be any integer value that satisfies consistency constraints (Equation 6). This means that if that relation is not involved in any fundamental cycle, then φ is chosen independently from the other relations (like in the case of the MP3 playback application, Section V). Indeed, it is sufficient to choose a value that minimizes the sum of capacities of channels between p and q. It is worth remembering that if a channel is going from q to p, then we have to reverse the affine relation.

In the following, we show how to compute the minimum size of a channel $c = (x, y)$ and the minimum number of its initial tokens assuming a complete $(n, \varphi, d)-$affine relation. Let us take $g_x = u_1(v_1)$ and $g_y = u_2(v_2)$.

Proposition 3. *The computation is limited to* $k = \frac{|v_1|\ |v_2|}{\gcd(n'|v_1|, d'|v_2|)}$ *instances of the affine relation pattern such that* $n' = \frac{n}{\gcd(n,d)}$ *and* $d' = \frac{d}{\gcd(n,d)}$.

Proof: We know that the minimum pattern in an affine relation consists of $d' = \frac{d}{\gcd(n,d)}$ ticks of $\hat{p}$ and $n' = \frac{n}{\gcd(n,d)}$ ticks of $\hat{q}$. To restrict the computation to a finite number of ticks, we have to repeat the minimum pattern in a coherent way with the amplitude functions; i.e. we must have $|v_1| \mid kd'$ and $|v_2| \mid kn'$.

So, $|v_1| \mid kd'$ implies that k is a multiple of $\frac{|v_1|}{\gcd(|v_1|, d')}$. In the same way, $|v_2| \mid kn'$ implies that k is a multiple of $\frac{|v_2|}{\gcd(|v_2|, n')}$. Therefore, the minimum value of k is equal to $\text{lcm}(\frac{|v_1|}{\gcd(|v_1|, d')}, \frac{|v_2|}{\gcd(|v_2|, n')}) = \frac{|v_1|\ |v_2|}{\gcd(n'|v_1|, d'|v_2|)}$. □

Example 2. In Figure 5, actors p and q are $(4, 5, 6)$-affine-related. The output port x is associated with $g_x = 3, 1(0, 1)$, while the input port y is associated with $g_y = 2(1, 1, 1, 0)$. The boundedness criterion is satisfied in this case. The computation is limited to $k = 2$ pattern instances of the affine relation as depicted in the figure. γ_1 is the number of tokens in the channel ($\bar{c}$ is assumed to be 0) w.r.t. the underflow worst-case scenario. If $\min \gamma_1 < 0$, then $\bar{c}$ must be equal to $-\min \gamma_1$ to guarantee that no underflow exception occurs during execution. γ_2 is the number of tokens in the channel ($\bar{c}$ is computed before) w.r.t. the overflow worst-case scenario. The minimum channel size is $\max \gamma_2$.

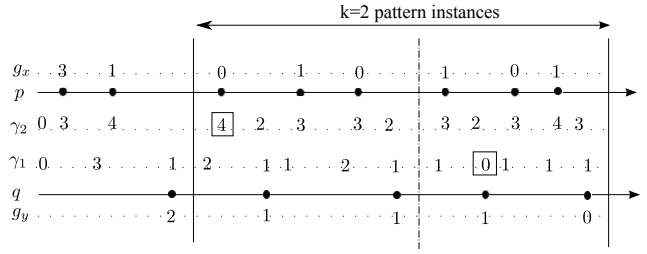

Figure 5. computation of the minimum size of a buffer and the minimum number of its initial tokens.

We can deduce from Equations 7 and 9 that $\lim_{|\varphi| \to \infty} \frac{h(c)}{|\varphi|} = \frac{||v_2||}{|v_2|d}$. This indicates that the best value of φ is in the neighborhood of 0.

An approximate solution: This solution consists in generating an integer linear program (ILP) for which the solution determines φ of every incomplete $(n, \varphi, d)-$affine relation, and $h(c)$ and $\bar{c}$ of every channel c.

For every channel $c = (x, y)$ between two actors p and q, we generate the following linear constraints. The first batch of generated constraints are $0 \leq \bar{c} \leq h(c)$ and $h(c) \geq \beta_x$, where β_x is the maximum element of g_x. If $h(c)$ and/or $\bar{c}$ are user-imposed, then we have to use constant values instead. Secondly, we generate two additional constraints, one for the overflow condition (Equation 7) and the other for the underflow condition (Equation 9).

Next, for every fundamental cycle in the graph of affine relations, we shall generate a linear constraint for its consistency condition (Equation 6).

Now, the objective function of the ILP is to minimize the sum of buffer sizes. We have to take care about cases where the sizes of tokens are different from one channel to another. If the linear program has a solution, then we are able to find all the *complete* affine relations. The channel sizes and the numbers of initial tokens obtained by the solution are not accurate. Therefore, we use the verification method described in the enumerative solution to compute the minimum size and the minimum number of initial tokens of each channel.

C. Step 3: Timing synthesis

While the previous step computes an abstract affine schedule of the data-flow graph, timing synthesis further aims at computing a concrete schedule. Indeed, the abstract schedule can be implemented as a static schedule or a dynamic one. Assuming a priority-driven preemptive scheduling policy, this step tries to define the period and the phase of each actor in a way that respects the affine relations and ensures schedulability.

Each actor p is mapped to a periodic task which has a period $\pi_p \geq \mathit{WCET}(p)$, a phase $\sigma_p \geq 0$, and a relative deadline equal to its period. Those timing characteristic are assumed to be *integers*.

If actors p and q are (n, φ, d)-affine-related, then the time concretization of the affine relation is given by the following linear constraints:

- $n\pi_q = d\pi_p$.
- if $\varphi \geq 0$ then $\sigma_q - \sigma_p = \frac{\varphi}{n}\pi_p$.
- if $\varphi < 0$ then $\sigma_p - \sigma_q = \frac{-\varphi}{d}\pi_q$.

In words, concretization of affine relations imposes constant time intervals between the ticks of every activation clock.

Let $\mathbb{G}$ be the connectivity graph of the data-flow network. $\mathbb{G}$ is an undirected graph where vertices are actors and edges represent affine relations. Those relations are computed based on the channel boundedness criterion (Steps 1/2), but can also be given by the designer, or deduced from user-imposed periods of actors (Step 1). First of all, we extract all the connected components of $\mathbb{G}$. For every actor p in a connected component $\mathbb{G}_i$, we have that $\pi_p = \frac{n_p}{d_p}\pi_i^*$ such that $n_p, d_p \in \mathbb{N}^+ | \gcd(n_p, d_p) = 1$ and π_i^* is the period of a fixed actor in $\mathbb{G}_i$. Since periods are integers, π_i^* must be a multiple of $\mathit{lcm}\{d_p | p \in \mathbb{G}_i\}$. In addition, a constraint like $\sigma_q - \sigma_p = \frac{\varphi}{n}\pi_p$ implies that $\frac{\varphi}{n}\pi_p \in \mathbb{N}$. In summary, π_i^*

must be a multiple of some integer m_i. From the worst-case execution times and bounds on periods, we compute a lower bound $\inf_i$ and an upper bound $\sup_i$ of π_i^*, respectively; i.e. $\inf_i \leq \pi_i^* \leq \sup_i$.

If the user imposes a period of an actor in $\mathbb{G}_i$, then π_i^* can be easily found, and it has to respect the previous constraints: i.e. $\inf_i \leq \pi_i^* = k * m_i \leq \sup_i$. In this case, we say that $\mathbb{G}_i$ is solved. There is at most one solved connected component. Algorithm 1 aims to solve the remaining components.

The most famous priority-driven scheduling algorithms are the EDF and the RM algorithms. Their scheduling analysis for a mono-processor system can be performed by just checking a schedulability condition. For a set of N periodic tasks, the *processor utilization factor* U is given by $\sum_p \frac{WCET(p)}{\pi_p}$. The task set is schedulable on one processor by EDF if and only if $U \leq 1$. It is schedulable by RM if $U \leq N(\sqrt[N]{2} - 1)$. The timing synthesis consists in finding timing characteristics so that $U \leq \alpha$ ($\alpha = 1$ for EDF, and $\alpha = N(\sqrt[N]{2} - 1)$ for RM).

For every connected component $\mathbb{G}_i$, we put $\psi_i = \sum_{p \in \mathbb{G}_i} \frac{WCET(p) d_p}{n_p}$. So, $U_i = \frac{\psi_i}{\pi_i^*}$ is the contribution of $\mathbb{G}_i$ to U. Algorithm 1 tries to find $\{\pi_i^* | \forall i\}$ which balance the contributions of components to U and maximize this latter.

Let S be the set of unsolved connected components ordered according to the decreasing order of m_i, and let $\mathbb{G}_0$ be the solved component (if any). Line 3 of the algorithm computes periods and phases of actors in $\mathbb{G}_0$ for the given π_0^* assuming that $U_0 \leq \alpha$, otherwise the task set is infeasible. The subsequent lines compute π_i^* of every unsolved connected component in such a way $U_i \leq avg = \frac{\alpha - U_0}{|S|}$. Let us put $\overline{U}_i$ to be the minimum contribution of $\mathbb{G}_i$ w.r.t. its sup_i. If $\overline{U}_i > avg$, then we are obliged to take π_i^* to be the maximum. Iterating over S in the decreasing order of m_i helps to increase the total U.

Algorithm 1 Timing Synthesis

Require: S: the set of ordered unsolved components. $\mathbb{G}_0$: the solved component (if any).
Ensure: the period and the phase of each actor.

```
1:  α = α − U_0;
2:  if α < 0 then the task set is infeasible; exit;
3:  Solve G_0;
4:  while S ≠ ∅ do
5:      avg = α/|S|;   b = true;
6:      for every G_i ∈ S do
7:          if Ū_i > avg then
8:              α = α − Ū_i;
9:              if α < 0 then the task set is infeasible; exit;
10:             Solve G_i w.r.t. the maximum of π_i^*;
11:             Remove G_i from S; b=false;
12:         end if
13:     end for
14:     if b then break;
15: end while
16: if S ≠ ∅ then
17:     k = |S|;
18:     for every G_i ∈ S do
19:         Solve G_i w.r.t. α/k;
20:         α = α − U_i;   k = k − 1;
21:     end for
22: end if
```

Figure 6. MP3 playback CSDF model.

V. Experimental validation

We now apply our algorithms to determine the periods and phases of actors and the buffer capacities and their number of initial tokens of an MP3 playback application. We consider the same MP3 playback CSDF model, depicted in Figure 6, that was used in many works [9]–[11]. Unlike these related works, we do not add reverse edges to model bounded buffers. Our algorithms can be applied on a CSDF model because this latter is a subclass of an unclocked ADF model. In the MP3 playback application, the MP3 task decodes a compressed audio stream to a 48 *kHz* audio sample stream which is next converted by the Sample Rate Converter (SRC) task to a 44.1 *kHz* stream. This stream is converted to an analog signal by the Digital-Analog Converter (DAC) task after its perceived quality is enhanced by the Audio Post-Processing (APP) task.

In both [10] and [11], the MP3 actor has five different firing functions (called phases) in correspondence with its amplitude function. Unlike with our algorithms, related techniques do not impose periodic releases of the actor but periodic releases of its phases. Therefore, we take *WCET*(MP3) to be equal to $\max$ *WCETs*(MP3) $= \max\{670, 2700, 720, 2700, 720\} = 2700\mu s$. Both the APP and DAC actors have a worst-case execution time equal to $22\mu s$. Table I shows the sum of the buffer sizes for different execution times of the SRC actor assuming that all tokens have the same size. `SumF` is the sum obtained when using the ILP synthesis of affine relations, while `SumE` is the sum obtained by the enumerative solution. For the MP3 playback application, the difference between `SumF` and `SumE` is 540 and which is acceptable w.r.t the numbers in the amplitude functions.

Confirmed by the results, we recal that the computed sizes do not depend on the worst-execution time of actors, unlike the results of [10] and [11]. Indeed, our analysis instead computes the affine relations between actors according to the amplitude functions associated with ports. Then, the channel sizes are induced from those affine relations which makes them independent from either the target machine

Table I
BUFFER CAPACITIES FOR THE MP3 PLAYBACK CSDF MODEL.

WCET(SRC) in *ms*	10	7.5	5	2.5
SumF	3152	3152	3152	3152
SumE	2612	2612	2612	2612
Sum (from [10])	2260	2054	1816	1578
Sum (from [11])	2228	2022	1816	1514

or the implementation code of actors. The implementation characteristics are included only in the timing synthesis step. As one can notice, `SumE` is worse than the sum obtained in related works. This was expected since actors can freely choose when they consume or produce tokens.

Table II
TIMING CHARACTERISTICS OF THE MP3 PLAYBACK TASKS IN *ms*.

	EDF		RM	
	π	σ	π	σ
MP3	13.219416	0	17.4636	0
SRC	27.54045	66.647889	36.3825	88.04565
APP	0.06245	121.760014	0.0825	160.8519
DAC	0.06245	121.916139	0.0825	161.05815

Table II shows the periods and phases of actors which satisfy either the EDF or the RM schedulability tests when WCET(SRC)=2.5 *ms*. The processor utilization factor U will be 99.96% in the EDF case and 75.66% in the RM case. It is not possible to schedule the application on one processor (using EDF or RM scheduler) and have the frequency of the digital-analog converter equal to $44.1kHz$ unless we use a more powerful processor.

A. Automatic SCJ code generation

This subsection describes how to synthesize a SCJ application from a data-flow specification. The user has to provide the Java code of firing functions in which the `set()` and `get()` methods are applied on ports in order to produce and consume one token, respectively. To perform our analysis, we first need to infer the amplitude function associated with each port from the Java code. This step is not yet implemented, therefore we suppose that amplitude functions are given as a part of the data-flow specification.

The SCJ application consists of one (so-called) mission. Every actor in the graph is implemented as a PEH registered to that mission. The `handleAsyncEvent()` methods are generated from the firing functions by substituting calls for `set()` and `get()` as described in the subsequent paragraph. For a fixed-priority SCJ scheduler, the timing parameters of PEHs are set as computed by the timing synthesis algorithm (RM case). Priorities of PEHs are set according to the rate monotonic politics, i.e. the shorter the period is, the higher is the actor's priority. Listing 1 shows a fragment of the SCJ code generated from the MP3 playback graph.

```
@Scope(IMMORTAL)
@SCJAllowed(value=LEVEL_1, members=true)
public class MP3Playback extends Mission {

    public byte[] C1= new byte[1824];  //MP3 --> SRC
    public byte[] C2= new byte[1324];  //SRC --> APP
    public byte[] C3= new byte[4];     //APP --> DAC

    @SCJRestricted(INITIALIZATION)
    protected void initialize() {
          /* initialize buffers if necessary */
        /* create & register PEHs (MP3, SRC, APP, DAC) */
  PeriodicParameters timing=new PeriodicParameters(new
       RelativeTime(0, 0), new RelativeTime(17, 463600));
  PeriodicEventHandler mp3 = new MP3("MP3",new
       PriorityParameters(11), timing,new
       StorageParameters(...));
          mp3.register();
  /* similarly for the others s.t. priority_SRC=10 and
       priority_APP=priority_DAC=12. */
    }
    public MissionSequencer getSequencer() {...}
    public void setUp() {...} ...
}

@SCJAllowed(value=LEVEL_1, members=true)
public class SRC extends PeriodicEventHandler {
       private int jc1=0;   private int ic2=0;

       public void handleAsyncEvent() {
           /* generated from the firing function code;
                a call to set(v) is replaced by: */
    Mission.getCurrentMission().C2[ic2]=v;
    ic2=(ic2+1)%1324;
       } ...
} ...
```

Listing 1. SCJ implementation of the MP3 playback data-flow model.

PEHs communicate through channels instantiated in the mission memory since each PEH has a private memory work space. A channel $c = (x, y)$ is implemented as a cyclic array C of a fixed size $h(c)$. The instruction `x.set(v)` will be substituted in the implementation by $\{$`C[ic]=v; ic=(ic+1)%h(c);` $\}$ such that `ic` is an additional local variable in the producer. Calls for the `get()` method are substituted in a similar way. Our analysis guarantees that neither an overflow nor an underflow exception will be thrown during execution, hence there is no need for synchronization protocols to access the array.

VI. CONCLUSION

Through a MoC based on activation clocks and affine relations, we have shown the necessary conditions for executions of data-flow graphs to be free of overflow and underflow exceptions over communication channels. We also presented an algorithm that, using integer linear programming, computes a symbolic affine schedule of a graph in a way that minimizes the buffering memory requirements and ensures the execution correctness. This schedule is independent from the target machine and the implementation code of actors. Unlike related works, we do not constraint actors to consume all their tokens before executing their firing functions and to write their results at the end. This choice led to a conservative analysis but gave more freedom when writing the Java implementation code of firing functions.

We presented a timing synthesis algorithm that concretizes the affine schedule, i.e. assigns a period and a phase to each actor so that the set of actors becomes schedulable on a uniprocessor system with an EDF or RM scheduler. The algorithm aims to maximize the processor utilization factor, and allows the user to impose upper bounds on periods or to define some of them.

Finally, we showed, through an example, how the SCJ implementation code of an application can be automatically generated from its dataflow specification. Future work will consider timing synthesis for multi-processor systems and attempt to increase the expressiveness of the ADF model.

REFERENCES

[1] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, pp. 1–34, March 2004.

[2] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: high-throughput stream programming in Java," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 211–228.

[3] J. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek, "Flexible task graphs: a unified restricted thread programming model for Java," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '08. New York, NY, USA: ACM, 2008, pp. 1–11.

[4] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.

[5] M. Geilen and T. Basten, "Requirements on the execution of Kahn process networks," in *Proceedings of the 12th European Conference on Programming*, ser. ESOP'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 319–334.

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, pp. 397–408, 1996.

[7] E. A. Lee and D. G. Messerchmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, pp. 24–35, January 1987.

[8] T. M. Parks, J. L. Pino, and E. A. Lee, "A comparaison of synchronous and cycle-static dataflow," in *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers*, ser. ASILOMAR'95, vol. 2. Washington, DC, USA: IEEE Computer Society, 1995, pp. 204–210.

[9] S. Stuijk, M. Geilen, and T. Basten, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," *IEEE Trans. Comput.*, vol. 57, pp. 1331–1345, October 2008.

[10] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Efficient computation of buffer capacities for cyclo-static datatflow graphs," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC '07. New York, NY, USA: ACM, 2007, pp. 658–663.

[11] M. Benazouz, O. Marchetti, A. M. Kordon, and T. Michel, "A new method for minimizing buffer sizes for cyclo-static dataflow graphs," in *ESTIMedia*. IEEE, 2010, pp. 11–20.

[12] S. Goddard and K. Jeffay, "Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application," in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, ser. RTAS' 97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 60–.

[13] K. Jeffay and D. Bennett, "A rate-based execution abstraction for multimedia computing," in *Proceedings of the 5th International Workshop on Network and Operating System Support*, ser. NOSSDAV '95. London, UK: Springer-Verlag, 1995, pp. 64–75.

[14] M. Bamakhrama and T. Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *Proceedings of the 9th ACM International Conference on Embedded Software*, ser. EMSOFT '11. New York, NY, USA: ACM, 2011, pp. 195–204.

[15] JSR-302, "Safety critical Java technology specification," October 2010.

[16] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek, "Developing safety critical java applications with oSCJ/L0," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '10. New York, NY, USA: ACM, 2010, pp. 95–101.

[17] E. A. Lee and E. Matsikoudis, "The semantics of dataflow with firing," in *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, 1st ed., G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, Eds. Cambridge University Press, 2009, ch. 4, pp. 71–94.

[18] I. M. Smarandache and P. L. Guernic, "Affine transformations in Signal and their application in the specification and validation of real-time systems," in *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software: Transformation-Based Reactive Systems Development*, ser. ARTS'97. London, UK: Springer-Verlag, 1997, pp. 233–247.