A Process Network Model for Reactive Streaming Software with Deterministic Task Parallelism

Fotios Gioulekas[1], Peter Poplavko[2], Panagiotis Katsaros[1,5(✉)], Saddek Bensalem[3], and Pedro Palomo[4]

[1] Aristotle University of Thessaloniki, Thessaloniki, Greece
{gioulekas,katsaros}@csd.auth.gr
[2] Mentor®, A Siemens Business, Montbonnot, France
petro.poplavko@siemens.com
[3] Université Grenoble Alpes (UGA), VERIMAG, Grenoble, France
Saddek.Bensalem@univ-grenoble-alpes.fr
[4] Deimos Space®, Madrid, Spain
pedro.palomo@deimos-space.com
[5] Information Technology Institute, Centre of Research and Technology,
Thessaloniki, Greece

Abstract. A formal semantics is introduced for a Process Network model, which combines streaming and reactive control processing with task parallelism properties suitable to exploit multi-cores. Applications that react to environment stimuli are implemented by communicating sporadic and periodic tasks, programmed independently from an execution platform. Two functionally equivalent semantics are defined, one for sequential execution and one real-time. The former ensures functional determinism by implying precedence constraints between jobs (task executions), hence, the program outputs are independent from the task scheduling. The latter specifies concurrent execution on a real-time platform, guaranteeing all model's constraints; it has been implemented in an executable formal specification language. The model's implementation runs on multi-core embedded systems, and supports integration of runtime managers for shared HW/SW resources (e.g. for controlling QoS, resource interference or power consumption). Finally, a model transformation approach has been developed, which allowed to port and statically schedule a real spacecraft on-board application on an industrial multi-core platform.

Keywords: Process network · Stream processing · Reactive control
Real-time

The research leading to these results has received funding from the European Space Agency project MoSaTT-CMP, Contract No. 4000111814/14/NL/MH.

A. Russo and A. Schürr (Eds.): FASE 2018, LNCS 10802, pp. 94–110, 2018.
https://doi.org/10.1007/978-3-319-89363-1_6

1 Introduction

The proliferation of multi-cores in timing-critical embedded systems requires a programming paradigm that addresses the challenge of ensuring predictable timing. Two prominent paradigms and a variety of associated languages are widely used today. For streaming signal processing, synchronous dataflow languages [18] allow writing programs in the form of directed graphs with nodes for their functions and arcs for the data flows between functions. Such programs can exploit concurrency when they are deployed to multi-cores [15], while their functions can be statically scheduled [17] to ensure a predictable timing behavior.

On the other hand, the reactive-control synchronous languages [12] are used for reactive systems (*e.g.,* flight control systems) expected to react to stimuli from the environment within strict time bounds. The synchronicity abstraction eliminates the non-determinism from the interleaving of concurrent behaviors.

The synchronous languages lack appropriate concepts for task parallelism and timing-predictable scheduling on multiprocessors, whereas the streaming models do not support reactive behavior. The *Fixed Priority Process Network* (FPPN) model of computation has been proposed as a trade-off between streaming and reactive control processing, for task parallel programs. In FPPNs, task invocations depend on a combination of periodic data availability (similar to streaming models) and sporadic control events. Static scheduling methods for FPPNs [20] have demonstrated a predictable timing on multi-cores. A first implementation of the model [22] in an executable formal specification language called BIP (Behavior, Interaction, Priority) exists, more specifically in its real-time dialect [3] extended to tasks [10]. In [21], the FPPN scheduling was studied by taking into account resource interference; an approach for incrementally plugging online schedulers for HW/SW resource sharing (*e.g.,* for QoS management) was proposed.

This article presents the first comprehensive FPPN semantics definition, at two levels: semantics for sequential execution, which ensures functional determinism, and a real-time semantics for concurrent task execution while adhering to the constraints of the former semantics. Our definition is related to a new model transformation framework, which enables programming at a high level by embedding FPPNs into the architecture description, and allows an incremental refinement in terms of task interactions and scheduling[1]. Our approach is demonstrated with a real spacecraft on-board application ported onto the European Space Agency's quad-core Next Generation Microprocessor (NGMP).

2 Related Work

Design frameworks for embedded applications, like Ptolemy II [6] and PeaCE [11], allow designing systems through refining high-level models. They are based on various models of computation (MoC), but we focus mainly on those that support task scheduling with timing constraints. Dataflow MoCs that

[1] The framework is online at [2].

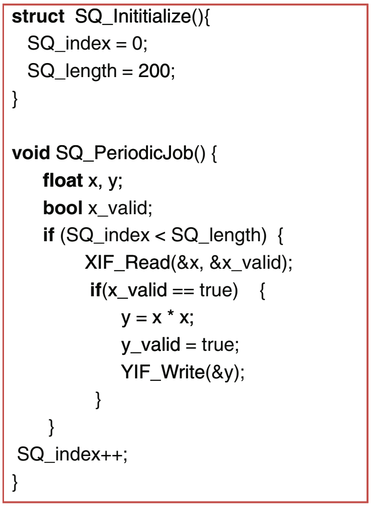
stem from the Kahn Process Networks [16] have been adapted for the timing constraints of signal processing applications and design frameworks like CompSoC [13] have been introduced; these MoCs do not support reactive behavior and sporadic tasks as in the FPPN MoC that can be seen as an extension in that direction. DOL Critical [10] ensures predictable timing, but its functional behavior depends on scheduling. Another timing-aware reactive MoC that does not guarantee functional determinism is the DPML [4]. The Prelude design framework [5] specifies applications in a synchronous reactive MoC, but due to its expressive power it is hard to derive scheduling analyses, unless restricting its semantics. Last but not the least, though the reactive process networks (RPN) [8] do not support scheduling with timing constraints, they lay an important foundation for combining the streaming and reactive control behaviors. In the FPPN semantics we reuse an important principle of RPN semantics, namely, performing the *maximal execution run* of a dataflow network in response to a control event.

3 A PN Model for Streaming and Reactive Control

An FPPN model is composed of *Processes*, *Data Channels* and *Event Generators*.

A *Process* represents a software subroutine that operates with internal variables and input/output channels connected to it through ports. The *functional code* of the application is defined in processes, whereas the necessary *middleware* elements of the FPPN are channels, event generators, and *functional priorities*, which define a relation between the processes to ensure deterministic execution.

An example process is shown in Fig. 1. This process performs a check on the internal variables, if the check succeeds then it reads from the input channel, and, if the value read *is valid* (refer to the channel definition below) its square is computed. The write operation on an output channel is then performed. A call to the process subroutine is referred to as a *job*. Like the real-time jobs, the subroutine should have a bounded execution time subject to WCET (worst-case execution time) analysis.



```
struct SQ_Inititialize(){
  SQ_index = 0;
  SQ_length = 200;
}

void SQ_PeriodicJob() {
    float x, y;
    bool x_valid;
    if (SQ_index < SQ_length)  {
         XIF_Read(&x, &x_valid);
         if(x_valid == true)    {
              y = x * x;
              y_valid = true;
              YIF_Write(&y);
         }
    }
 SQ_index++;
}
```

Fig. 1. Example code for "Square" process

An FPPN is defined by two directed graphs. The first is a (possibly cyclic) graph (P, C), whose nodes P are processes and edges C are channels for pairs of communicating processes with a dataflow direction, *i.e.,* from the writer to the reader (there are also external channels interacting with the environment).

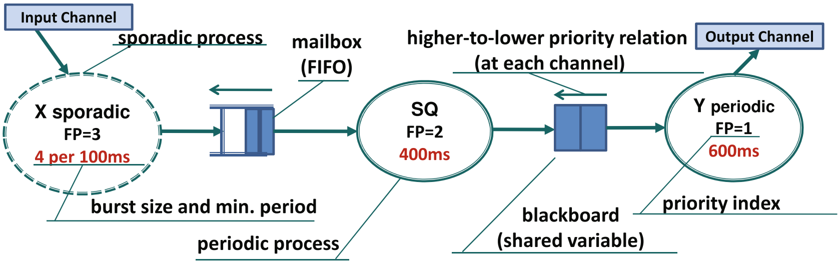


Fig. 2. Example Fixed Priority Process Network

A channel is denoted by a $c \in C$ or a pair (p_1, p_2) of writer and reader. For p_1 the channel is said to be an output and for p_2 an input. The second graph $(P, \mathcal{FP})$ is the functional priority directed acyclic graph (DAG) defining a functional priority relation between processes. For any two communicating processes we require,

$$(p_1, p_2) \in C \implies (p_1, p_2) \in \mathcal{FP} \vee (p_2, p_1) \in \mathcal{FP}$$

i.e., a functional priority either follows the direction of dataflow or the opposite. Given a $(p_1, p_2) \in \mathcal{FP}$, p_1 is said to have a *higher priority* than p_2.

The FPPN in Fig. 2, represents an imaginary data processing application, where the “X” sporadic process generates values, “Square” calculates the square of the received value and the “Y” periodic process serves as sink for the squared value. A sporadic event (command from the environment) invokes “X”, which is annotated by its minimal inter-arrival time. The periodic processes are annotated by their periods. The two types of non-blocking channels are also illustrated. The FIFO (or mailbox) has a semantics of a queue. The blackboard remembers the last written value that can be read multiple times. The arc depicted above the channels indicates the functional priority relation $\mathcal{FP}$. Additionally, the external input/output channels are shown. In this example, the dataflow in the channels go in the opposite direction of the functional priority order. Note that, by analogy to the scheduling priorities, a convenient method to define priority is to assign a unique priority index to every process, the smaller the index the higher the priority. This method is demonstrated in Fig. 2. In this case the minimal required $\mathcal{FP}$ relation would be defined by joining each pair of communicating processes by an arc going from the higher-priority process to the lower-priority one.

Let us denote by *Var* the set of all variables. For a variable x or an ordered set (vector) X of variables we denote by $\mathbf{D}(x)$ (resp. $\mathbf{D}(X)$) its domain (or vector of domains), *i.e.,* the set(s) of values that the variable(s) may take. Valuations of variables X are shown as $X^0, X^1 \ldots$, or simply as X, dropping the superscript. Each variable is assumed to have a unique initial valuation. From the software point of view, this means that all variables are initialized by a default value.

Var includes all *process state* variables X_p and the *channel state* variables γ_c. The current valuation of a state variable is often referred to simply as *state.*

For a variable of channel c, an alphabet Σ_c and a type CT_c are defined; a *channel type* consists of write 'operations' (W_c) and read 'operations' (R_c) defined as functions specifying the variable evolution. Function $W_c : \mathbf{D}(c) \times \Sigma_c \rightarrow \mathbf{D}(c)$ defines the update after writing a symbol $s \in \Sigma_c$ to the channel, whereas $R_c : \mathbf{D}(c) \rightarrow \mathbf{D}(c) \times \Sigma_c$ maps the channel state to a pair (Rc_1, Rc_2), where Rc_1 is the new channel state and Rc_2 is the symbol that is read from the channel. For a FIFO channel, its state γ_c is a (initially empty) string and the write operation left-concatenates symbol s to the string: $W_c(\gamma_c, s) = s \circ \gamma_c$. For the same channel, $R_c(\gamma_c \circ s) = (\gamma_c, s)$, *i.e.,* we read and remove the last symbol from the string. The write and read functions are defined for each possible channel state, thus rendering the channels non-blocking. This is implemented by including $\perp$ in the alphabet, in order to define the read operation when the channel does not contain any 'meaningful' data. Thus, reading from an empty FIFO is defined by: $R_c(\epsilon) = (\epsilon, \perp)$, where ϵ denotes an empty string. For blackboard channel, its state is a (initially empty) string that contains at most one symbol – the last symbol written to the channel: $W_c(\gamma_c, s) = s$, $R_c(\gamma_c) = (\gamma_c, \gamma_c)$, $R_c(\epsilon) = (\epsilon, \perp)$.

An *external channel*'s state is an infinite sequence of samples, *i.e.,* variables $c[1], c[2], c[3], \ldots$ with the same domain. For a sample $c[k]$, k is the *sample index.* Though the sequence is infinite, no infinite memory is required, because each sample can be accessed (as will be shown) within a limited time interval. If c is an external output, the channel type defines the sample write operation in the form $W'_c : \mathbf{D}'(c) \times \mathbb{N}_+ \times \Sigma_c \rightarrow \mathbf{D}'(c)$, where $\mathbf{D}'(c)$ is the sample domain, the second argument is the sample index and the result is the new sample value. For an external input, we have the sample read operation $R_c : \mathbf{D}'(c) \times \mathbb{N}_+ \rightarrow \mathbf{D}'(c) \times \Sigma_c$. The set of outputs is denoted by O and the set of inputs by I.

The program expressions involve variables. Let us call *Act* the set of all possible *actions* that represent operations on variables. An assignment is an action written as $Y := f(X)$. For the channels, two types of actions are defined, $x!c$ for writing a variable x, and $x?c$ for reading from the channel, where $\mathbf{D}(x) = \Sigma_c$. For external channels, we have $x!_{[k]}c,\ c \in O$ and $y?_{[k]}c,\ c \in I$, where $[k]$ is the sample index. Actions are defined by a function *Effect* : $Act \times \mathbf{D}(Var) \rightarrow \mathbf{D}(Var)$, which for every action a states how the new values of all variables are calculated from their previous values. The *actions are assumed to have zero delay.* The physical time is modeled by a special action for waiting until time stamp τ, $\mathbf{w}(\tau)$.

An *execution trace* $\alpha \in Act^*$ is a sequence of actions, *e.g.,*

$$\alpha = \mathbf{w}(0),\ x?_{[1]}I_1,\ x := x^2,\ x!c_1, \mathbf{w}(100),\ y?c_1,\ O_1!_{[2]}y$$

The time stamps in the execution are non-decreasing, and denote the time until the next time stamp, at which the following actions occur. In the example, at time 0 we read sample [1] from I_1 and we compute its square. Then we write to channel c_1. At time 100, we read from c_1 and write the sample [2] to O_1.

A process models a subroutine with a set of locations (code line numbers), variables (data) and operators that define a guard on variables ('if' condition), the action (operator body) and the transfer of control to the next location.

Definition 1 (Process). *Each process p is associated with a deterministic transition system ($\ell_p{}^0$, L_p, X_p, $X_p{}^0$, $\mathcal{I}_p$, $\mathcal{O}_p$, A_p, $\mathcal{T}_p$), with L_p a set of locations, $\ell_p{}^0 \in L_p$ an initial location, and X_p the set of state variables with initial values $X_p{}^0$. $\mathcal{I}_p, \mathcal{O}_p$ are (internal and external) input/output channels. A_p is a set actions with variable assignments for X_p, reads from $\mathcal{I}_p$, and writes to $\mathcal{O}_p$. $\mathcal{T}_p$ is transition relation $\mathcal{T}_p : L_p \times G_p \times A_p \times L_p$, where G_p is the set of predicates (guarding conditions) defined on the variables from X_p.*

One *execution step* $(\ell_1, X^1, \gamma^1) \overset{g:a}{\to} (\ell_2, X^2, \gamma^2)$ for the valuations X^1, X^2 of variables in X_p and the valuations γ^1, γ^2 of channels in $\mathcal{I}_p \cup \mathcal{O}_p$, implies that there is transition $(\ell_1, g, a, \ell_2) \in \mathcal{T}_p$, such that X^1 satisfies guarding condition g (*i.e.*, $g(X^1) = True$) and $(X^2, \gamma^2) = \mathit{Effect}(a, (X^1, \gamma^1))$.

Definition 1 prescribes a deterministic transition system: for each location ℓ_1 the guarding conditions enable for each possible valuation X^i a single execution step.

Definition 2 (Process job execution). *A job execution $(X^1, \gamma^1) \xrightarrow{\alpha}_p (X^2, \gamma^2)$ is a non-empty sequence of process p execution steps starting and ending in p's initial location ℓ_0, without intermediate occurrences of ℓ^0:*

$$(\ell^0, X^1, \gamma^1) \overset{g_1:\alpha_1}{\to} (\ell_1, X_1, \gamma_1) \ldots \overset{g_n:\alpha_n}{\to} (\ell^0, X^2, \gamma^2), \; \textit{for } n \geq 1, \ell_i \neq \ell^0$$

From a software point of view, a job execution is seen as a subroutine run from a caller location that returns control back to the caller. We assume that at k-th job execution, external channels I_p, O_p are read/written at sample index $[k]$.

In an FPPN, there is a one-to-one mapping between every process p and the respective event generator e that defines the constraints of interaction with the environment. Every e is associated with (possibly empty) subsets I_e, O_e of the external input/output (I/O) channels. Those are the external channels that the process p can access: $I_e \subseteq \mathcal{I}_p$, $O_e \subseteq \mathcal{O}_p$. The I/O sets of different event generators are disjoint, so different processes cannot share external channels.

Every e defines the set of possible sequences of time stamps τ_k for the 'event' of k-th invocation of process p and a relative deadline $d_e \in \mathbb{Q}_+$. The intervals $[\tau_k, \tau_k + d_e]$ determine when the k-th job execution can occur. This timing constraint has two important reasons. First, if the subsets I_e or O_e are not empty then these intervals should indicate the timing windows when the environment opens the k-th sample in the external I/O channels for read or write access at the k-th job execution. Secondly, τ_k defines the order in which the k-th job should execute, the earlier it is invoked the earlier it should execute. Concerning the τ_k sequences, two event generator types are considered, namely *multi-periodic* and *sporadic*. Both are parameterized by a burst size m_e and a period T_e. Bursts of m_e periodic events occur at 0, T_e, $2T_e$, etc. For sporadic events, at most m_e events can occur in any half-closed interval of length T_e. In the sequel we associate the attributes of an event generator with the corresponding process, *e.g.*, T_p and d_p.

Definition 3 (FPPN). *An* FPPN *is a tuple* $\mathcal{PN} = (P, C, \mathcal{FP}, e_p, I_e, O_e, d_e, \Sigma_c, CT_c)$*, where* P *is a set of processes and* $C \subseteq P \times P$ *is a set of internal channels, with* (P, C) *defining a (possibly cyclic) directed graph. An acyclic directed graph* $(P, \mathcal{FP})$ *is also defined, with* $\mathcal{FP} \subset P \times P$ *a functional priority relation (if* $(p_1, p_2) \in \mathcal{FP}$*, we also write* $p_1 \rightarrow p_2$*). This relation should be defined at least for processes accessing the same channel,* i.e., $(p_1, p_2) \in C \Rightarrow p_1 \rightarrow p_2 \vee p_2 \rightarrow p_1$. e_p *maps every process* p *to a unique event generator, whereas* I_e *and* O_e *map each event generator to (possibly empty) partitions of the global set of external input channels* I *and output channels* O*, resp.* d_e *defines the relative deadline for accessing the I/O channels of generator* e*,* Σ_c *defines alphabets for internal and external I/O channels and* CT_c *specifies the channel types.*

The priority $\mathcal{FP}$ defines the order in which two processes are executed *when invoked at the same time.* It is not necessarily a transitive relation. For example, if $(p_1, p_2) \in \mathcal{FP}$, $(p_2, p_3) \in \mathcal{FP}$, and both p_1 and p_3 get invoked simultaneously then $\mathcal{FP}$ does not imply any execution-order constraint between them unless p_2 is also invoked at the same time. The functional priorities differ from the scheduling priorities. The former disambiguate the order of read/write accesses to internal channels, whereas the latter ensure satisfaction of timing constraints.

4 Zero-Delay Semantics for the FPPN Model

The functional determinism requirement prescribes that the data sequences and time stamps at the outputs are a well-defined function of the data sequences and time stamps at the inputs. This is ensured by the so-called functional priorities. In essence, functional priorities control the process job execution order, which is equivalent to the effect of fixed priorities on a set of tasks under uniprocessor fixed-priority scheduling with zero task execution times. A distinct feature of the FPPN model is that priorities are not used directly in scheduling, but rather in the definition of model's semantics. From now on, the term 'task' will refer to an FPPN process. Following the usual real-time systems terminology, invoking a task implies generation of a job which has to be executed before the task's deadline. The so-called *precedence constraints*, *i.e.,* the semantical restrictions of FPPN job execution order are implied firstly from the time stamps when the tasks are invoked and secondly from the functional priorities. In this section, we define these constraints in terms of a sequential order (an execution trace).

The FPPN model requires that *all simultaneous process invocations should be signaled synchronously.* This can be realized by introducing a periodic clock with sufficiently small period (the *gcd* of all T_p), such that invocations events can only occur at clock ticks, synchronously. Two variant semantics are then defined, namely the *zero-delay* and the *real-time* semantics.

The *zero-delay semantics* imposes an ordering of the job executions assuming that they have zero delay and that they are never postponed to the future. Since in this case the deadlines are always met even without exploiting parallelism, a sequential execution of processes is considered for simplicity. The semantics is defined in terms of the rules for constructing the execution trace of the FPPN for a given sequence $(t_1, \mathbf{P}^1)$, $(t_2, \mathbf{P}^2)$ $\ldots$, where $t_1 < t_2 < \ldots$ are time stamps and $\mathbf{P}^i$ is the multiset of processes invoked at time t_i. For convenience, we associate each 'invoked process' p in $\mathbf{P}^i$ with respective invocation event, e_p. The execution trace has the form:

$$Trace(\mathcal{PN}) = \mathbf{w}(t_1) \circ \alpha^1 \circ \mathbf{w}(t_2) \circ \alpha^2 \ldots$$

where α^i is a concatenation of job executions of processes in $\mathbf{P}^i$ included in an order, such that if $p_1 \rightarrow p_2$ then the job(s) of p_1 execute earlier than those of p_2.

Definition 4 (Configuration). *An FPPN configuration* $(\pi, \gamma, \mathbf{P})$ *consists of:*

- *a process configuration* π*, a function that assigns to every process a state* $\pi(p) \in \mathbf{D}(X_p)$
- *a channel configuration* γ*,* i.e., *the states of internal and external channels*
- *a set of pending events* $\mathbf{P}$

Executing one job in a process network:

$$\frac{\begin{array}{c}(\pi(p), \gamma) \xrightarrow{\alpha}_p (X', \gamma') \wedge e_p \in \mathbf{P} \\ \wedge \\ \nexists p' : e_{p'} \in \mathbf{P} \wedge (p', p) \in \mathcal{FP}\end{array}}{(\pi, \gamma, \mathbf{P}) \xrightarrow{\alpha}_{\mathcal{PN}} (\pi\{X'/p\}, \gamma', \mathbf{P} \setminus \{e_p\})}$$

where $\pi\{X'/p\}$ is obtained from π by replacing the state of p by X'.

Given a non-empty set of events $\mathbf{P}$ invoked at time t, a *maximal execution run* of a process network is defined by a sequence of job executions that continues until the set of pending events is empty.

$$\frac{(\pi^0, \gamma^0, \mathbf{P}) \xrightarrow{\alpha_1}_{\mathcal{PN}} (\pi_1, \gamma_1, \mathbf{P} \setminus \{e_{p_1}\}) \xrightarrow{\alpha_2}_{\mathcal{PN}} \ldots (\pi^1, \gamma^1, \emptyset)}{(\pi^0, \gamma^0) \overset{\mathbf{w}(t) \circ \alpha_1 \circ \alpha_2 \circ \ldots}{\longmapsto}_{\mathcal{PN}(\mathbf{P})} (\pi^1, \gamma^1)}$$

Given an initial configuration (π^0, γ^0) and a sequence $(t_1, \mathbf{P}^1)$, $(t_2, \mathbf{P}^2)$ $\ldots$ of events invoked at times $t_1 < t_2 < \ldots$, the run of process network is defined by a sequence of maximal runs that occur at the specified time stamps.

$$Run(\mathcal{PN}) = (\pi^0, \gamma^0) \overset{\alpha^1}{\longmapsto}_{\mathcal{PN}(\mathbf{P}^1)} (\pi^1, \gamma^1) \overset{\alpha^2}{\longmapsto}_{\mathcal{PN}(\mathbf{P}^2)} \cdots$$

The execution trace of a process network is a projection of the process network run to actions:

$$Trace(\mathcal{PN}) = \alpha^1 \circ \alpha^2 \ldots$$

This trace represents the time stamps $(\mathbf{w}(t_1), \mathbf{w}(t_2) \ldots)$ and the data processing actions executed at every time stamp. From the effect of these actions it is possible to determine the sequence of values written to the internal and external channels. These values depend on the states of the processes and internal channels. The concurrent activities – the job executions – that modify each process/channel states are deterministic themselves and are ordered relatively to each other in a way which is completely determined by the time stamps and the $\mathcal{FP}$ relation. Therefore we can make the following claim.

Proposition 1 (Functional determinism). *The sequences of values written at all external and internal channels are functionally dependent on the time stamps of the event generators and on the data samples at the external inputs.*

Basically, this property means that the outputs calculated by FPPN depend only on the event invocation times and the input data sequences, but not on the scheduling. To exploit task parallelism, in the real-time semantics of Sect. 5 the sequential order of execution and the zero-delay assumption are relaxed.

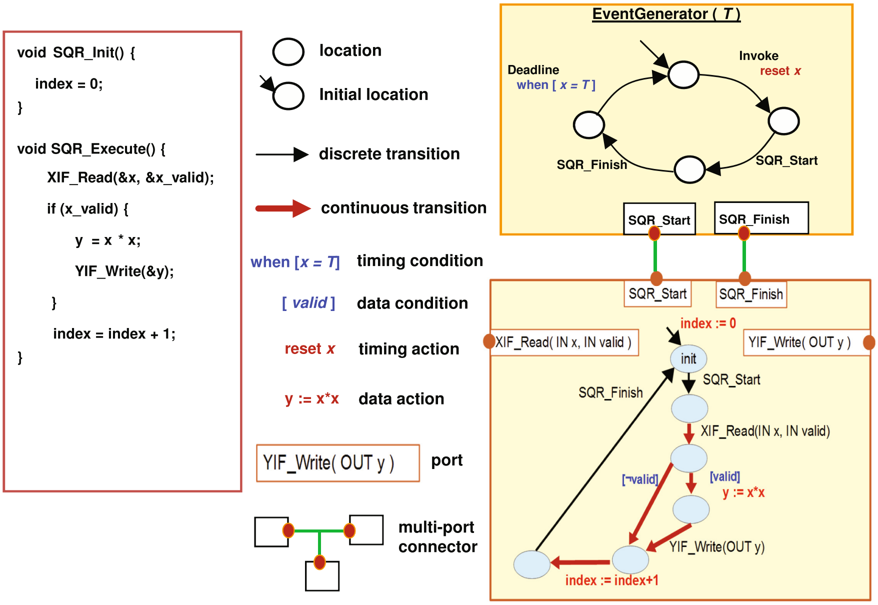
5 Real-Time Semantics for the FPPN Model

In the real-time semantics, job executions last for some physical time and can start concurrently with each other at any time after their invocation. Certain precedence constraints are respected which for certain jobs impose the same relative order of execution as in the zero-delay semantics, so that non-deterministic updates of the states of processes and channels are excluded. To ensure timeliness, the jobs should complete their execution within the deadline after their invocation. The semantics specifies the entities for communication, synchronization, scheduling and is defined by compilation to an *executable* formal specification language.

Our approach is based on (real-time) 'BIP' [3] for modeling networks of connected timed automata components [24]. We adopt the extension in [10], which introduces the concept of *continuous* (asynchronous) automata transitions, which, unlike the default (discrete) transitions take a certain physical time. Next to support of tasks (via continuous transitions), BIP supports the urgency in timing constraints, and those are timed-automata features required for adequate modeling and timing verification of dataflow languages [9]. An important BIP language feature for implementing the functional code of tasks is the possibility to specify data actions in imperative programming language (C/C++).

Figure 3 illustrates how an FPPN process is compiled to a BIP component. The source code is parsed, searching for primitives that are relevant for the interactions of the process with other components. The relevant primitives are the reads and writes from/to the data channels. For those primitives the generated BIP component gets ports, *e.g.,* 'XIF_Read(IN x,IN valid)', through which the respective transitions inside the component synchronize and exchange data with other components. In line with Definition 1, every job execution corresponds to a sequence of transitions that starts and ends in an initial location. The first transition in this sequence, 'Start', is synchronized with the event generator component, which enables this transition only after the process has been invoked. The event generator shown in Fig. 3 is a simplified variant for periodic tasks whose deadline is equal to the period. In [22] it is also described how we model internal channels and give more details on event generator modelling.

To ensure a functional behavior equivalent to zero-delay semantics, the job executions have to satisfy precedence constraints between subsequent jobs of the same process, and the jobs of process pairs connected by a channel. In both



```
void SQR_Init() {
   index = 0;
}

void SQR_Execute() {
     XIF_Read(&x, &x_valid);
     if (x_valid) {
          y  = x * x;
          YIF_Write(&y);
      }
      index = index + 1;
}
```

Fig. 3. Compilation of functional code to BIP

cases, the relative execution order of these subsets of jobs is dictated by zero-delay semantics, whereby the jobs are executed in the invocation order and the simultaneously invoked jobs follow the functional priority order. In this way, we ensure deterministic updates in both cases: (i) for the states of processes by *excluding auto-concurrency*, and (ii) for the data shared between the processes by *excluding data races* on the channels. The precedence constraints for (i) are satisfied by construction, because BIP components for processes never start a new job execution until the previous job of the same process has finished. For the precedence constraints in (ii), an appropriate component is generated for each pair of communicating processes and plugged incrementally into the network of BIP components.

Figure 4 shows such a component generated a given pair of processes “A” and “B”, assuming (A, B) $\in \mathcal{FP}$. We saw in Fig. 3 that the evolution of a job execution goes through three steps: ‘invoke’, ‘start’ and ‘finish’. The component handles the three steps of both processes in almost symmetrical way, except in the method that determines whether the job is ready to start: if two jobs are simultaneously invoked, then first the job of process “A” gets ready and then, after it has executed, the job of “B” becomes ready. The “Functional Priority”

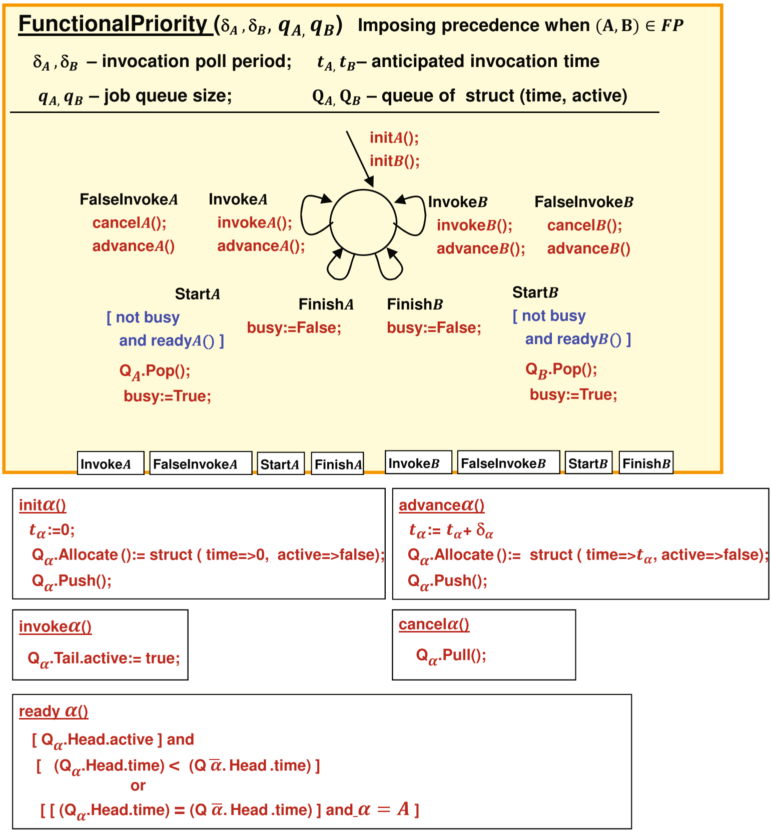

FunctionalPriority (δ_A, δ_B, q_A, q_B) Imposing precedence when $(A, B) \in FP$
δ_A, δ_B – invocation poll period; t_A, t_B– anticipated invocation time
q_A, q_B – job queue size; Q_A, Q_B – queue of struct (time, active)
init$A$();
init$B$();
FalseInvoke$A$
cancel$A$();
advance$A$()
Invoke$A$
invoke$A$();
advance$A$();
Invoke$B$
invoke$B$();
advance$B$();
FalseInvoke$B$
cancel$B$();
advance$B$()
Start$A$
[not busy
and ready$A$()]
Q_A.Pop();
busy:=True;
Finish$A$
busy:=False;
Finish$B$
busy:=False;
Start$B$
[not busy
and ready$B$()]
Q_B.Pop();
busy:=True;
Invoke$A$
FalseInvoke$A$
Start$A$
Finish$A$
Invoke$B$
FalseInvoke$B$
Start$B$
Finish$B$
initα()
t_α:=0;
Q_α.Allocate ():= struct (time=>0, active=>false);
Q_α.Push();
advanceα()
t_α:= t_α+ δ_α
Q_α.Allocate ():= struct (time=>t_α, active=>false);
Q_α.Push();
invokeα()
Q_α.Tail.active:= true;
cancelα()
Q_α.Pull();
ready α()
[Q_α.Head.active] and
[(Q_α.Head.time) < (Q $\overline{\alpha}$. Head .time)]
or
[[(Q_α.Head.time) = (Q $\overline{\alpha}$. Head .time)] and $\alpha = A$]

Fig. 4. Imposing precedence order between "A", "B" ("A" has higher functional priority)

component maintains two job queues[2] denoted Q_α where $\alpha \in \{A, B\}$ indicates a process selection. In our notation, $\overline{\alpha}$ means 'other than α', *i.e.*, if $\alpha = A$ then $\overline{\alpha} = B$ and if $\alpha = B$ then $\overline{\alpha} = A$.

The component receives from the event generator of process 'α' at regular intervals with period δ_α either 'Invoke α' or 'FalseInvoke α'. In the latter case (*i.e.*, no invocation), the job in the tail of the queue is 'pulled' away[3].

[2] Queues are implemented by a circular buffer with the following operations:
- `Allocate`() picks an available (statically allocated) cell and gives reference to it
- `Push`() push the last allocated cell into the *tail*
- `Pull`() undo the push
- `Pop`() retrieve the data from the *head* of the queue.

[3] Thanks to 'init α' and 'advance α', the queue tail always contains the next anticipated job, which is conservatively marked as non-active until 'Invoke α' transition.

6 Model Transformation Framework

The model-based design philosophy for embedded systems which we follow [14] is grounded on the evolutionary design using models, which support the gradual refinement (refined models are more accurate than those refined) and the setting of real-time attributes that ensure predictable timing. Such a process allows considering various design scenarios and promotes the late binding to design decisions. Our approach to refinement is based on *incremental component-based* models, where the system is evolved by incrementally plugging new components and transforming existing ones.

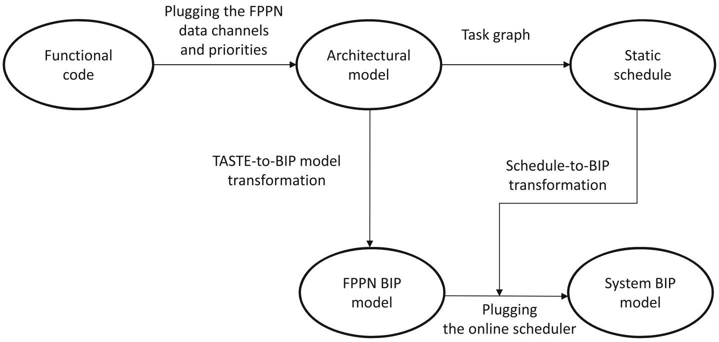

Functional code
Plugging the FPPN data channels and priorities
Architectural model
Task graph
Static schedule
TASTE-to-BIP model transformation
Schedule-to-BIP transformation
FPPN BIP model
Plugging the online scheduler
System BIP model

Fig. 5. Evolutionary design of time-critical systems using FPPNs

We propose such a design approach (Fig. 5), in which we take as a starting point a set of tasks defined by their *functional code* and real-time attributes (*e.g.,* periods, deadlines, WCET, job queue capacity). We assume that these tasks are encapsulated into software-architecture functional blocks, corresponding to FPPN processes. Before being integrated into a single *architectural model* they can be compiled and tested separately by functional simulation or by running on embedded platform.

The high-level architecture description framework of our choice is the TASTE toolset [14,19], whose front-end tools are based on the AADL (Architecture Analysis & Design Language) syntax [7]. An *architecture model* in TASTE consists of functional blocks – so-called 'functions' – which interact with each other via pairs of interfaces (IF) 'required IF'/'provided IF', where the first performs a procedure call in the second one. In TASTE, the provided interfaces can be explicitly used for task invocations, *i.e.,* they may get attributes like 'periodic'/'sporadic', 'deadline' and 'period'. The FPPN processes are represented by TASTE 'functions' that 'provide' such interfaces, implementing job execution of the respective task in C/C++. Our TASTE-to-BIP framework is available for download at [2].

The first refinement step is plugging the data channels for explicit communication between the processes. The data channels are also modeled as TASTE functions, whereas reads and writes are implemented via interfaces. We have

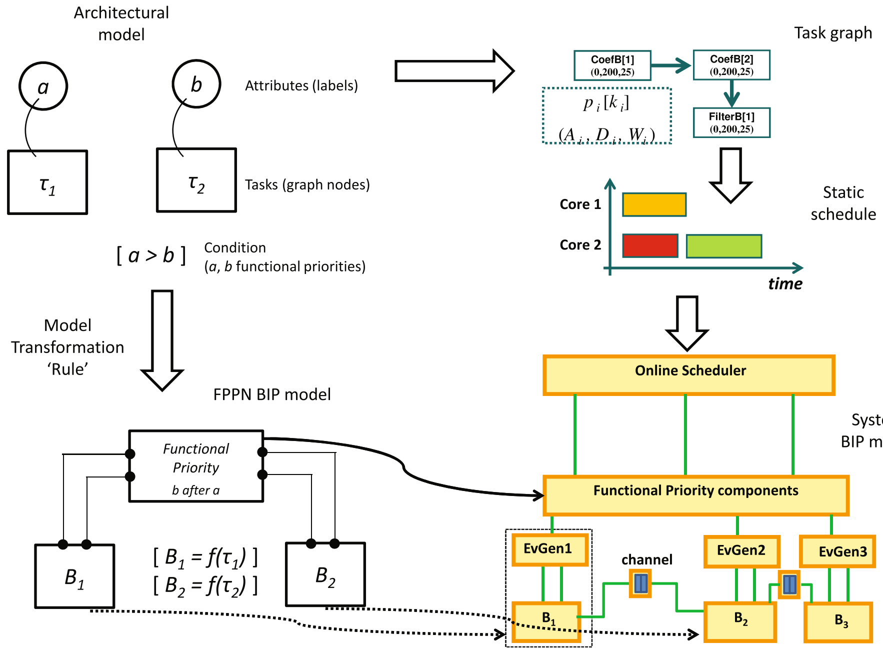


Fig. 6. Model and graph transformations for the FPPN semantics

amended the attributes of TASTE functions to reflect the priority index of processes and the parameters of FPPN channels, such as capacity of FIFO channels. The resulting model can be compiled and simulated in TASTE.

The second and final refinement step is scheduling. To schedule on multi-cores while respecting the real-time semantics of FPPN this step is preceded by transformation from TASTE architectural model into BIP FPPN model. The transformation process implements the FPPN-to-BIP 'compilation' sketched in the previous section, and we believe it could be formalized by a set of *transformation rules*. For example, as illustrated in Fig. 6, one of the rules could say that if there are two tasks τ_1 and τ_2 related by $\mathcal{FP}$ relation then their respective BIP components B_1 and B_2 are connected (via 'Start' and 'Finish' ports) to a functional priority component.

The scheduling is done offline, by first deriving a task graph from the architectural model, taking into account the periods, functional priorities and WCET of processes. The task graph represents a maximal set of jobs invoked in a hyperperiod and their precedence constraints; it defines the invocation and the deadline of jobs relatively to the hyperperiod start time. The task graph derivation algorithm is detailed in [20].

Definition 5 (Task Graph). *A directed acyclic graph $\mathcal{TG}(\mathcal{J}, \mathcal{E})$ whose nodes $\mathcal{J} = \{J_i\}$ are jobs defined by tuples $J_i = (p_i, k_i, A_i, D_i, W_i)$, where p_i is the job's process, k_i is the job's invocation count, $A_i \in \mathbb{Q}_{\geq 0}$ is the invocation time, $D_i \in \mathbb{Q}_+$ is the absolute deadline and $W_i \in \mathbb{Q}_+$ is the WCET. The k-th job of process p is denoted by $p[k]$. The edges $\mathcal{E}$ represent the precedence constraints.*

The task graph is given as input to a static scheduler. The schedule obtained from the static scheduler is translated into parameters for the *online-scheduler* (cf. Fig. 6), which, on top of the functional priority components, further constraints the job execution order and timing, with the purpose of ensuring deadline satisfaction. The joint application/scheduler BIP model is called System Model. This model is eventually compiled and linked with the BIP-RTE, which ensures correct BIP semantics of all components online [23].

7 Case Study: Guidance, Navigation and Control Application

Our design flow was applied to a Guidance Navigation & Control (GNC) onboard spacecraft application that was ported onto ESA's NGMP, more specifically the quad-core LEON4FT processor [1]. In the space industry, multi-cores provide a means for integrating more software functions onto a single platform, which contributes to reducing size, weight, cost, and power consumption. Onboard software has to efficiently utilize the processor resources, while retaining predictability.

A GNC application affects the movement of the vehicle by reading the sensors and controlling the actuators. We estimated the WCETs of all tasks, W_p, by measurements. There are four tasks: the Guidance Navigation Task ($T_p = 500$ ms, $d_p = 500$ ms, $W_p = 22$ ms), the Control Output Task ($T_p = 50$ ms, $d_p = 50$ ms, $W_p = 3$ms) that sends the outputs to the appropriate spacecraft unit, the Control FM Task ($T_p = 50$ ms, $d_p = 50$ ms, $W_p = 8$ ms) which runs the control and flight management algorithms, and the Data Input Dispatcher Task ($T_p = 50$ ms, $d_p = 50$ ms, $W_p = 6$ ms), which reads, decodes and dispatches data to the right destination whenever new data from the spacecraft's sensors are available. The hyperperiod of the system was therefore 500 ms, and it includes one execution of the Guidance Navigation Task and ten executions of each other task, which results in 31 jobs. The Guidance Navigation and Control Output tasks were invoked with relative time offsets 450 ms and 30 ms, respectively. Fig. 7 shows the GNC FPPN, where the functional priorities impose precedence from the numerically smaller FP index (*i.e.,* higher-priority) to the numerically larger ones, we defined them based on analysis of the specification documents and the original implementation of task interactions by inter-thread signalling.

The architectural model in TASTE format was automatically transformed into a BIP model and the task-graph model of the hyperperiod was derived. The task graph was passed to the static scheduler, which calculated the system load to be 112% (*i.e.,* at least two cores required, taking into account precedences [20] and interference [21]) and generated the static schedule.

The BIP model was compiled and linked with the BIP RTE and the executables were loaded and ran on the LEON4FT board. Figure 8 shows the measured Gantt chart of a hyper-period (500 ms) plus 100 ms. We label the process executions as 'P<id>', where '<id>' is a numeric process identifier. Label 'P20' is an exception, it indicates the execution of the BIP RTE engine and all discrete-event controllers – event generators, functional priority controllers, and the online

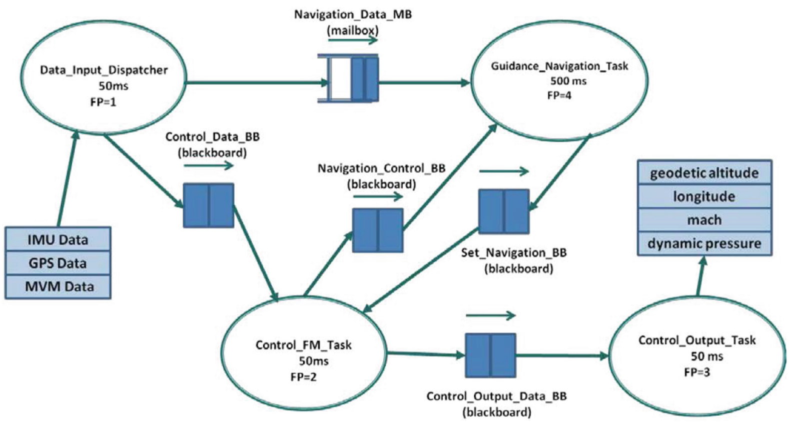


Fig. 7. The GNC FPPN model

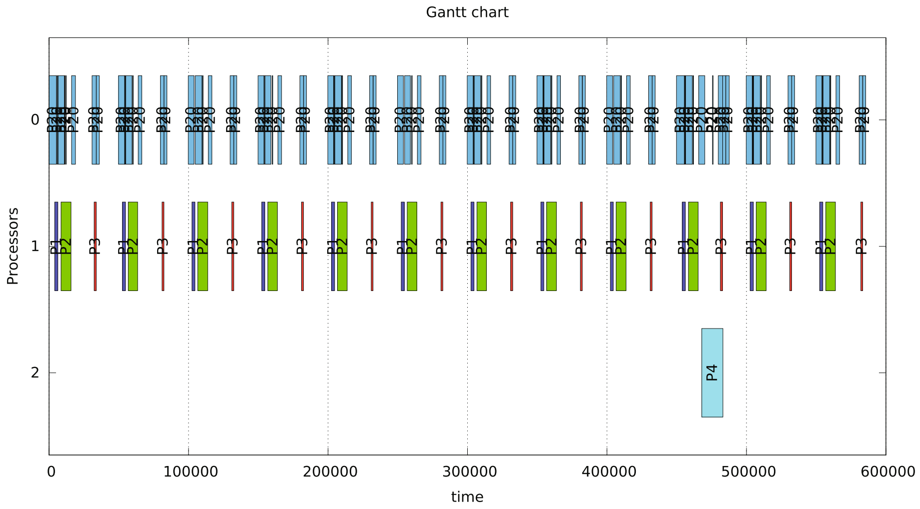


Fig. 8. Execution of the GNC application on LEON4FT (in microseconds).

scheduler. Since there are four discrete transitions per one job execution and 31 jobs per hyperperiod, 31 × 4 = 124 discrete transitions are executed by BIP RTE per hyperperiod. The P20 activities were mapped to Core 0, whereas the jobs of tasks (P1, P2, P3, P4) were mapped to Core 1 and Core 2. P1 stands for the Data Input Dispatcher, P2 for the Control FM, P3 for the Control Output and P4 for the Guidance Navigation task. Right after 10 consecutive jobs of P1, P2, P3 the job on P4 is executed. The job of P4 is delayed due to the 450 ms invocation offset and the least functional priority. Since P3 and P4 do not communicate via the channels, in our framework $(P3, P4) \notin \mathcal{FP}$ and they can execute in parallel, which was actually programmed in our static schedule. Due to more than 100% system load this was necessary for deadline satisfaction.

8 Conclusion

We presented the formal semantics of the FPPN model, at two levels: zero-delay semantics with precedence constraints on the job execution order to ensure functional determinism, and real-time semantics for scheduling. The semantics was implemented by a model transformational framework. Our approach was validated through a spacecraft on-board application running on a multi-core. In future work we consider it important to improve the efficiency of code generation, formal proofs of equivalence of the scheduling constraints (like the task graph) and the generated BIP model. The offline and online schedulers need to be enhanced to a wider spectrum of online policies and a better awareness of resource interference.

References

1. GR-CPCI-LEON4-N2X: Quad-core LEON4 next generation microprocessor evaluation board. http://www.gaisler.com/index.php/products/boards/gr-cpci-leon4-n2x
2. Multicore code generation for time-critical applications. http://www-verimag.imag.fr/Multicore-Time-Critical-Code,470.html
3. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: EMSOFT 2010 (2010)
4. Chaki, S., Kyle, D.: DMPL: programming and verifying distributed mixed-synchrony and mixed-critical software. Technical report, Carnegie Mellon University (2016). http://www.andrew.cmu.edu/user/schaki/misc/dmpl-extended.pdf
5. Cordovilla, M., Boniol, F., Forget, J., Noulard, E., Pagetti, C.: Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In: RTNS (2011)
6. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
7. Feiler, P., Gluch, D., Hudak, J.: The architecture analysis & design language (AADL): an introduction. Technical report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2006). http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879
8. Geilen, M., Basten, T.: Reactive process networks. In: EMSOFT 2004, pp. 137–146. ACM (2004)
9. Ghamarian, A.H.: Timing analysis of synchronous dataflow graphs. Ph.D. thesis, Eindhoven University of Technology (2008)
10. Giannopoulou, G., Poplavko, P., Socci, D., Huang, P., Stoimenov, N., Bourgos, P., Thiele, L., Bozga, M., Bensalem, S., Girbal, S., Faugere, M., Soulat, R., Dinechin, B.D.d.: DOL-BIP-Critical: a tool chain for rigorous design and implementation of mixed-criticality multi-core systems. Technical report (2016)
11. Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., Joo, Y.P.: PeaCE: a hardware-software codesign environment for multimedia embedded systems. ACM Trans. Des. Autom. Electron. Syst. **12**(3), 24:1–24:25 (2008)
12. Halbwachs, N.: Synchronous Programming of Reactive Systems. Springer, Berlin (2010). https://doi.org/10.1007/978-1-4757-2231-4

13. Hansson, A., Goossens, K., Bekooij, M., Huisken, J.: CoMPSoC: a template for composable and predictable multi-processor system on chips. ACM Trans. Des. Autom. Electron. Syst. (TODAES) **14**(1), 2 (2009)
14. Hugues, J., Zalila, B., Pautet, L., Kordon, F.: From the prototype to the final embedded system using the Ocarina AADL tool suite. ACM Trans. Embed. Comput. Syst. **7**(4), 42:1–42:25 (2008)
15. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**(1), 1–34 (2004)
16. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York (1974)
17. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **C–36**(1), 24–35 (1987)
18. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
19. Perrotin, M., Conquet, E., Delange, J., Schiele, A., Tsiodras, T.: TASTE: a real-time software engineering tool-chain overview, status, and future. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 26–37. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25264-8_4
20. Poplavko, P., Socci, D., Bourgos, P., Bensalem, S., Bozga, M.: Models for deterministic execution of real-time multiprocessor applications. In: DATE 2015, pp. 1665–1670. IEEE, March 2015
21. Poplavko, P., Kahil, R., Socci, D., Bensalem, S., Bozga, M.: Mixed-critical systems design with coarse-grained multi-core interference. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 605–621. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_42
22. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: A timed-automata based middleware for time-critical multicore applications. In: SEUS 2015, pp. 1–8. IEEE (2015)
23. Triki, A., Combaz, J., Bensalem, S., Sifakis, J.: Model-based implementation of parallel real-time systems. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 235–249. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37057-1_18
24. Waez, M.T.B., Dingel, J., Rudie, K.: A survey of timed automata for the development of real-time systems. Comput. Sci. Rev. **9**, 1–26 (2013)