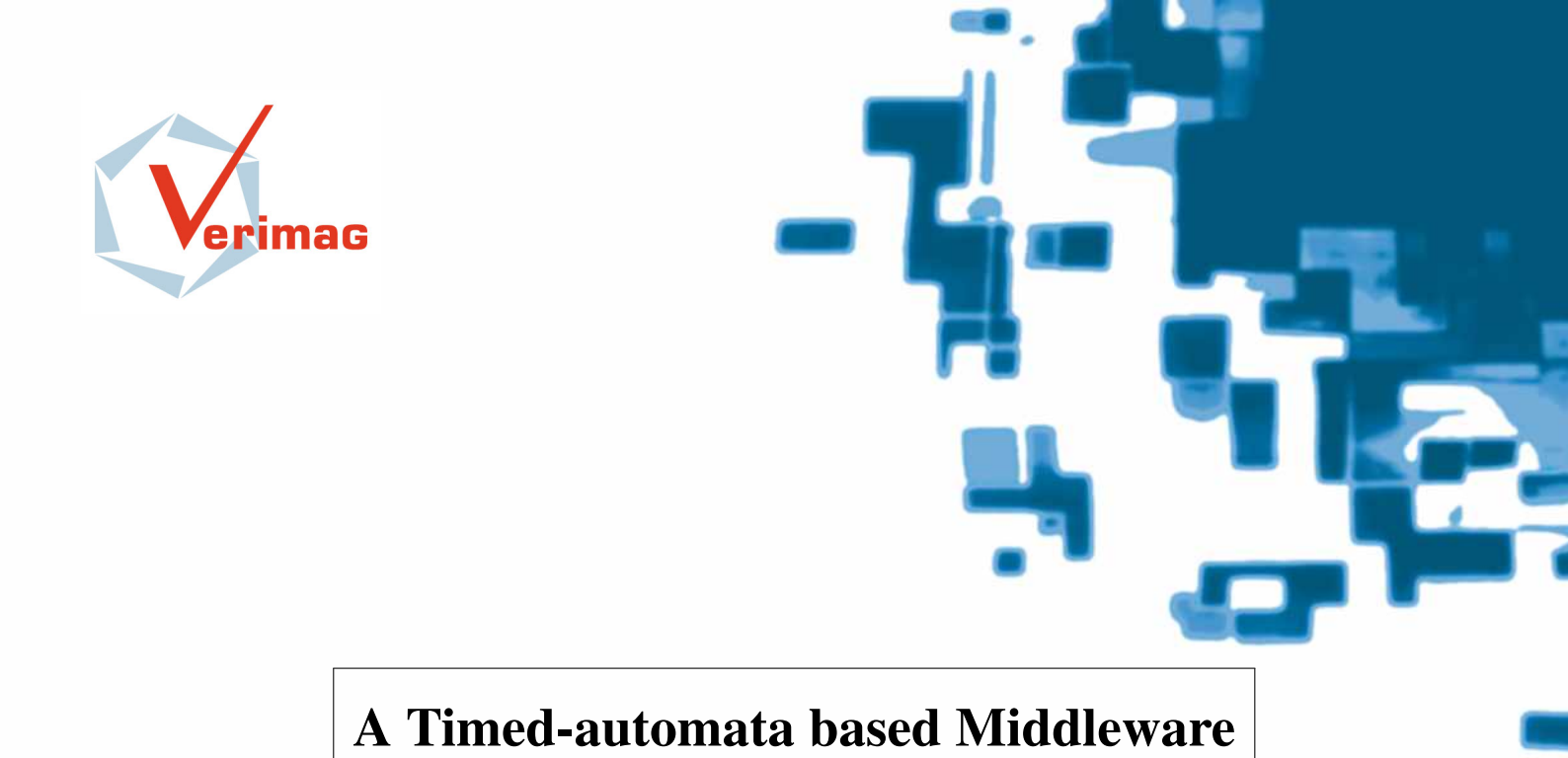

A Timed-automata based Middleware for Time-critical Multicore Applications

Dario Socci, Peter Poplavko, Paraskevas Bourgos, Saddek Bensalem, Marius Bozga

Verimag Research Report n^o TR-2015-12

December 2015

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

A Timed-automata based Middleware for Time-critical Multicore Applications[1][2]

Dario Socci, Peter Poplavko, Paraskevas Bourgos, Saddek Bensalem, Marius Bozga

December 2015

Abstract

Various models of computation for multi-core time-critical systems have been proposed in the literature, but there is a significant gap between the models of computation and the real-time scheduling and analysis techniques, that makes timing validation challenging. To overcome this difficulty, we represent both the models of computation and the scheduling policies by timed automata. While, traditionally, they are only used for simulation and validation, we use the automata for programming. We believe that using the same formal language for the model of computation and the scheduling techniques is an important step to close the gap between them. Our approach is demonstrated using a publicly available toolset, an industrial application use case and a multi-core platform.

Keywords: runtime environment, code generation, multi-cores, hard real-time systems, timing constraints, timing-critical systems, mixed-critical systems

Reviewers:

How to cite this report:

```
@techreport {TR-2015-12,
  title = {A Timed-automata based Middleware for Time-critical Multicore Applications},
   author = {Dario Socci, Peter Poplavko, Paraskevas Bourgos, Saddek Bensalem, Marius
Bozga},
  institution = {{Verimag} Research Report},
  number = {TR-2015-12},
  year = {2015}
}
```

[1] This report is an extended version of [1]

[2] The research leading to these results has received funding from **CERTAINTY** – European Community's Seventh Framework Programme [FP7/2007-2013], grant agreement no. 288175.

1 Introduction

The formal techniques help to address different challenges in real-time system design. One of them is lack of consolidation in programming. Embedded software design has in common with hardware design that it has to satisfy not only functional, but also extra-functional requirements, first of all, timing. However, unlike hardware languages, the software languages have an important deficiency: they were conceived without any concern on timing in mind [2]. Real-time programming is a very heterogeneous area of research, as it employs many different models of computation (MoCs), such as synchronous languages, timed Petri nets, various extensions of synchronous dataflow (SDF), etc. Expressing the software design in a given MoC is difficult, but, worse still, even when this is done, the real-time scheduling and timing analysis still remains challenging, due to a gap between the MoCs and the real-time scheduling policies [3].

Therefore it takes a lot of effort to compose a *middleware* by combining a particular model and particular policy. This task could be simplified if there existed a common 'backbone' language expressive enough to redefine and reuse different components of middleware. Partly, this idea was implemented in the SystemC project, offering a common way to express scheduling policies, MoCs and functional code[4]. However, this language lacks a formal semantics, and it mainly offers facilities for simulation only, but not for fully-automated deployment of software.

Therefore, as an alternative, we propose to use *combined procedural and automata languages*. To demonstrate the concept, we offer public prototype tools [5] for multicore timing-critical system design based on the timed-automata language RT-BIP [6]. This paper describes the design flow for programming and implementing timing-critical systems in this language. Section 2 gives an overview of the proposed design flow, Section 3 gives an introduction into RT-BIP, then in Section 4 we show how to translate a high-level description of the software, including the middleware, into RT-BIP. In Section 5 we describe implementation of an industrial use case on Kalray MPPA256 multi-core platform and some experimental results. Finally, Section 6 concludes the paper.

2 Background

In this section we present our proposed design flow. It is based on a combined procedural and timed-automata based language, referred to as *backbone language*. Examples of such languages are TIMES [7], IF [8], and RT-BIP [6]. A backbone language can be used for modeling, validation and simulation, but our main point is to use it as a programming language. Because in many cases, an automata-based language may be too low-level for direct use in application programming, we can *compile* higher-level models into the backbone language automatically. In the ideal case, this is ensured by letting the user create a set of rules for automatic translation of the functional code into the automata. Also the user would provide a set of automata templates that implement the primitives of the preferred MoC and scheduling policy. Hence the backbone language would serve as a meta-model and meta-policy used to program the desired timing-critical systems middleware. The specified set of rules and templates would allow to compile the functional code and the middleware into a network of timed automata that can be analyzed and deployed on a platform.

This idea is partly implemented in our design flow, see Fig. 1. The design flow accepts a high-level specification of application tasks (the MoC and the functional code) at the input and compiles it into the backbone language, for which we use RT-BIP [6]. Also the flow takes from the offline scheduling tool the specification of the online scheduling policy and the selected scheduling parameters (such as priorities) of the tasks. The scheduler is also compiled into backbone language and 'plugged' into the common RT-BIP software model. This model is deployed on the platform on top of the RT-BIP run-time environment (RTE) for multi-cores. The software model can also be combined with the hardware model to represent the complete software-hardware system and to perform timing analysis for validation of schedulability properties, but the validation part of the design flow is beyond the scope of the prototype toolset [5] and this paper.

Currently we support only one MoC – Fixed Priority Process Networks (FPPN) [9], which combines the abilities to model both the reactive-control and streaming applications. As for the scheduling policies, we support a combined static-order/time-triggered policy [10, 9]. In future we consider to provide means

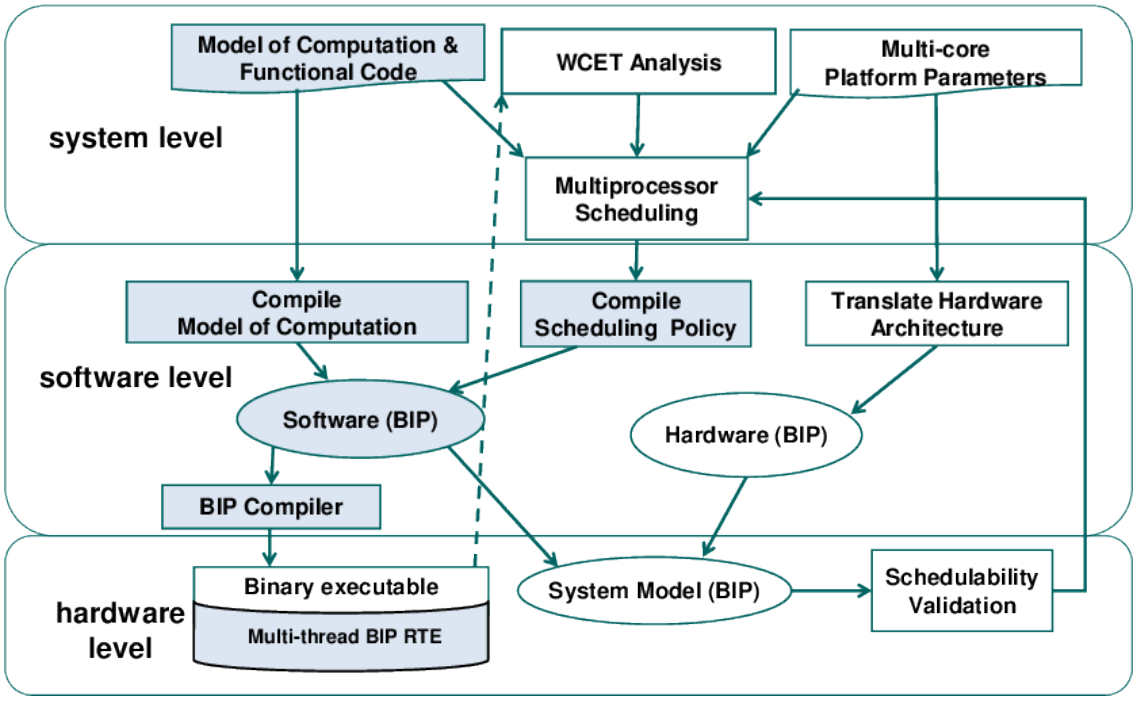

Figure 1: Design flow (highlighting the steps covered in this paper)

to the user to specify templates for his preferred MoC and policy. We also consider to add support for other relevant MoCs, such as synchronous languages, as in the Prelude [11] framework, and SDF, such as in CompSoC [12].

3 Backbone Language: Real Time BIP

Our backbone language is RT-BIP [6]. The "RT" prefix stands for *real-time*, which indicates that this language models the physical time, which is done using the same concept of clocks as in timed automata [13]. In fact, the RT-BIP language can be seen as a language to express networks of communicating timed automata that employs a specific compositional syntax and a specific flavor of timing constraints. The "BIP" part of the acronym stands for behaviour-interactions-priority. BIP fits our needs thanks to the concepts of *eager transitions* (also known as *urgent transitions*) and *rendezvous* (explained later in this section) that makes modeling easier compared to other timed automata based languages.

In this section, we give an overview of the particular dialect of RT-BIP that we use in our framework for timing-critical multi-core applications. Mainly, this dialect is a restriction of RT-BIP. We also add to the standard RT-BIP a new concept – the *continuous transitions*, which, unlike the standard RT-BIP transitions, are non-instantaneous. Fig. 2 shows an RT-BIP example that represents two tasks, A and B, running on two CPUs. The model consists of four components, namely, "PeriodicA", "DelayableB", "CPU1" and "CPU2". All the components are defined by an automaton and a set of ports.

The states of the BIP components are usually referred to as locations. A transition is an execution step from one location to itself or to another location. For example "(Skip)" is a transition from "S1" to "S0" in component "DelayableB".

Each transition has an associated enabling condition and an associated action that is executed at this transition. In our figures we show the conditions in blue color and in square brackets, e.g. condition $[D_{OUT} \neq 0]$ for transition "StartB" in "DelayableB". In BIP, every component is seen as an object in object-oriented programming sense. Every component encapsulates some data and some 'methods' (*i.e.,* subroutines) to manipulate the data. All actions executed by transitions can execute methods written in an imperative language (C/C++). The methods have access only to the local variables of the component

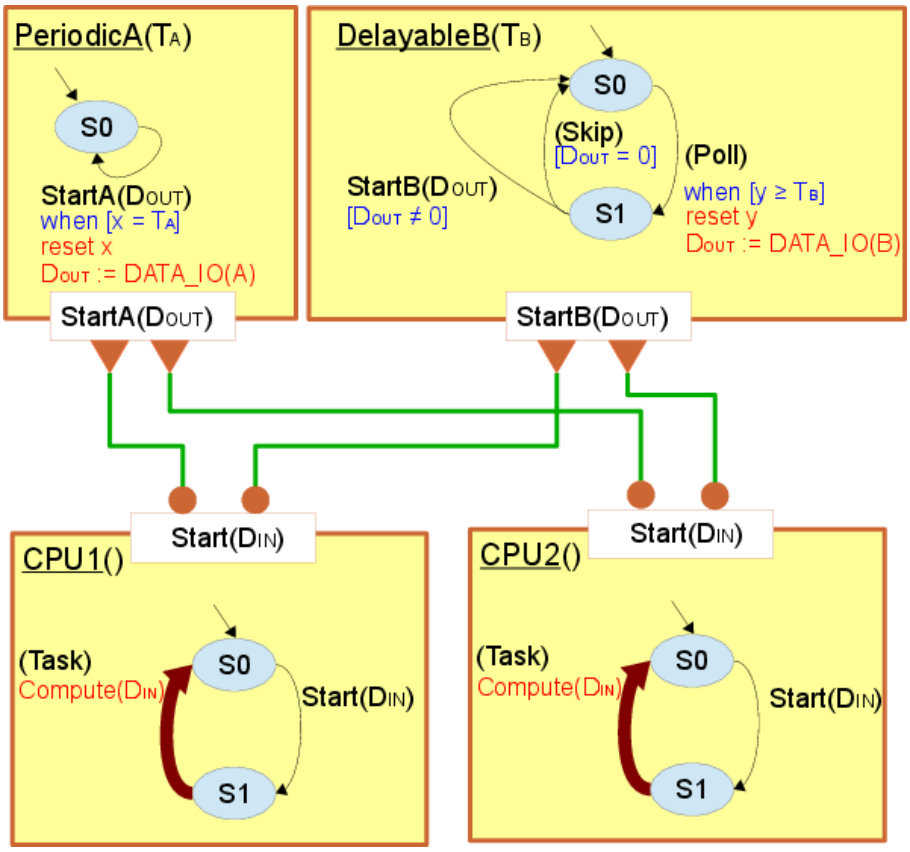

Figure 2: RT-BIP model example

itself, the components do not share variables. In the figures the actions are shown as blocks of pseudo-code in dark-red color.

Multiple components run concurrently and execute interactions with each other, the set of possible interactions being defined using connectors (shown in green lines) which join several ports (enclosed in white rectangles in our figures), which belong to components. In our RT-BIP dialect we use only one type of BIP connector, "rendezvous", which obliges to participate in an interaction *all ports connected to it* simultaneously. A port participating in interaction means that a transition annotated by the given port gets executed. This can only happen when the transition is enabled, *i.e.,* the automaton current state is the source location of the transition and all conditions associated to the transition are satisfied. For example, port "StartB" can participate in interaction only when the automaton is in the "S1" location and "$D_{OUT} \neq 0$". A port may participate in one interaction at a time. If in our example all four ports are enabled then four interactions can potentially occur, but only two of them will be *selected* (in non-deterministic way).

A transition can be either internal or external. Internal transitions are not annotated by ports and do not participate in interactions with other components. We enclose their annotations in parentheses to distinguish them, e.g. "(Task)" in our example. External transitions are annotated with a port, such as transition "StartB" in "DelayableB". The same port can be annotated on multiple transitions. An external transition executes simultaneously with an interaction at the corresponding port. Note that in our figures we sometimes annotate a port by a circle (*e.g.,* "Start(D_{IN})") or a triangle (*e.g.,* "StartB(D_{OUT}")). These annotations do not refer to any construct in the BIP language but indicate one of the two roles that the port typically plays in the interactions by construction of the model. The triangle means a "master" port and the circle means a "slave" port. A master joined by a connector to several slaves indicates that the slaves are required to be enabled whenever the master may get enabled. This requirement is satisfied by properly setting the timing condition, *i.e.,* in Fig. 2, the execution time of the continuous transition "(Task)" must be such that it never exceed T_A. This means that it is the master who "takes the initiative" for the interaction and at the moment when it is ready to execute the whole interaction can execute immediately. The master and slave notations in our figures serve only to improve the readability of the figures.

Each component has an initial transition that is executed at start of execution, shown as an arrow without a source location pointing to one of the locations, which makes it the initial location, such as location "S0" in "DelayableB". Each component has an associated set of private local variables: data variables and clocks. In line with the usual conventions adopted in timed automata, the clocks are real-valued variables that are initialized to zero at start and whose values are continuously increasing with the passage of physical time. In our models we use letters x, y for the clocks, e.g. the model in Fig. 2 uses two clocks. A clock can be reset to zero at a transition (e.g. "reset x" in "PeriodicA") and used in a condition, in which case the condition is preceded by the word "when".

We use only urgent timing conditions, referred to as 'eager' in RT-BIP. All transitions are eager in our model, which means that they start executing at the earliest moment of time when they are enabled for execution. For an external transition this means the earliest moment of time when all ports participating in an interaction with its port are enabled. For example, consider condition "when $[y \geq T_B]$" in Fig. 2. Due to this condition the interactions involving port "StartB" should wait for the earliest time when clock y reaches value at least T_B.

Though the transitions *start* urgently, another issue is when they *finish*. The instantaneous transitions take zero time (conceptually). By default, all RT-BIP transitions are instantaneous, with exception of those that have attribute 'continuous'. On contrary to instantaneous transitions, continuous ones take exactly the time required to execute the corresponding action, which can be any timing duration not known at the moment when the transition starts. In our figures we denote continuous transitions by thick arrows, *e.g.,* "(Task)" transitions in our model. In modeling the real-time tasks, we use such transitions to represent the blocks of execution where the task does not interact with the runtime environment but is performing internal computations instead, *e.g.,* running the "Compute()" method in our example in Fig. 2.

As for the data variables, in our examples we most often use three types: integer (such as D_{IN}, D_{OUT} in our example), Boolean and queue, though, of course, other types can be defined in RT-BIP using C/C++ syntax. Unless explicitly done otherwise in the action of the initial transition, we assume that the initial transition implicitly initializes the data variables to zero in the case of integers, "False" in the case of Booleans, and a type-specific initial value (such as empty queue) for other types. Next to data variables there are also compile-time parameters, for example, period T_A and minimal execution interval T_B in

Fig. 2.

Some variables are used for communication between the components at the interactions. Their values are sent and received via ports, therefore they are listed as port parameters. We assume that if a port has parameter $< name >_{OUT}$ then an interaction of this port assigns the value of this variable to the parameters $< name >_{IN}$ with the same $< name >$ of the other ports that participate in the interaction. For example, port $Start(D_{IN})$ receives the value of D_{IN} from the D_{OUT} of either "PeriodicA" or "DelayableB".

In our RT-BIP programs for time-critical systems we often use *queues*. This well-known data structure can be easily implemented using a circular buffer. We define the following operations on the queue:

- **Allocate**() returns the available buffer cell that is next to be pushed
- **Push**() pushes the 'allocated' cell into the *tail* of the queue
- **Pull**() undoes the push
- **Pop**() extracts the *head* of the queue

4 Compiling the Software into RT-BIP

The time-critical software consists of functional code and middleware, the latter providing elements for communication, synchronization and real-time scheduling. Compiling means translating the functional code and middleware specification into components of RT-BIP language and connecting them with each other. The components express the correct timing behavior by timing constraints at the transitions. A situation where for a component automaton no transitions are possible anymore in future is called local deadlock and is detected as a runtime error. The RT-BIP components generated at compilation are constructed in such a way that a deadlock indicates that either the hardware resources cannot handle the workload on time or that the workload does not conform to specification. For example, in Fig. 2, component "PeriodicA" is ready to execute an interaction at port "StartA" only when $x = T_A$. If at this moment of time both "CPU"components are busy executing the previously started "(Task)" transitions, then component "PeriodicA" will deadlock as the clock x will continue counting the time, never coming back to T_A anymore. To avoid a deadlock in "PeriodicA", at least one of the "CPU" components should be ready for interaction at periodic instances in time: $T_A, 2T_A, 3T_A, \ldots$. Similar conditions hold for certain BIP components generated at compilation.

4.1 The FPPN Model of Computation

In our framework, we currently work with a functionally-deterministic MoC intended to support both the reactive control and the streaming applications, the so-called *Fixed Priority Process Networks* (FPPN) [9]. An instance of FPPN is composed of three main entities: *Processes*, *Data Channels* and *Event Generators*. The determinism is ensured by *Functional Priority* relation between the processes.

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 FPPN are channels, event generators, and priorities.

An example of process is given in Fig. 3. This process example performs a check on the internal variables, then (if the check is successful) reads from the input channel, and, if the value is valid (see channel description below) computes the square of it. Then the write operation on an output channel is performed. A single invocation of the subroutine that defines the process is referred to as as a *job*. Like the real-time jobs, this subroutine should have a bounded execution time and is subject to WCET (worst-case execution time) analysis.

Data Channels ensure *read and write operations* for communication. These operations are *non-blocking*, which means that reading from an empty channel will not block the reader. Therefore, next to the data value, the read operation returns the so-called *validity flag*, *i.e.,* a Boolean indicator of whether the data is valid. There are inter-process and external (environment) channels. In this paper we consider only the inter-process channels. We define two channel types: FIFO and blackboard. Other types can be introduced by extension of the library of RT-BIP components. The FIFO has a semantics of a queue. The blackboard remembers the last written value that can be read multiple times. Reading from an empty FIFO or a non-initialized blackboard returns a validity flag set to 'false'.

```
struct Square::internal_var {
   int index = 0;
   int length = 200;
}

void Square::Job(internal_var *X) {
  float x,y;
  if ( X->index  <  X->length ) {
     read(PORT_IN, &x);
     if (PORT_IN.valid) {
              y  = x * x;
              write(PORT_OUT, &y);
     }
  }
  X->index = X->index + 1;
}
```

Figure 3: Functional Code for "Square" Process Example

An *event generator* e is defined by the set of possible sequences of time stamps τ_k that it can produce. We define two types of event generators: *periodic* and *sporadic*. *Every event generator is associated with a unique process* and determines whether the given process is periodic or sporadic one. Every process p has a deadline d_p. Interval $[\tau_k, \tau_k + d_p)$ determines the time interval when the k-th process job can be executed. At τ_k the job gets 'activated' and then it remains active until it is scheduled. After being scheduled, the job should terminate before the deadline. Periodic processes are activated at period T_p, for sporadic processes T_p denotes the minimal inter-arrival time. We define the *job queue length* as $q_p = \lceil d_p/T_p \rceil$, this quantity is the maximum number of jobs of process p that can be active simultaneously.

An FPPN network can be described by two directed graphs. The first graph is the default process network graph (P, C), whose nodes are processes P and the edges are channels C. This graph can be cyclic and defines the communicating pairs of processes and the direction of dataflow: from writer to reader. The second graph is the functional priority DAG: (P, FP). No cyclic paths are allowed in this graph. The edges define functional priority relation between the processes. It is however, not a partial order relation, as it is not necessarily transitive. We require that any two communicating processes have a priority relation:

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

i.e., a functional priority should either follow the direction of the data flow or the opposite direction.

Fig. 4 below gives an example of a process network. This process network represents an imaginary signal processing application with input sample period $200ms$, reconfigurable filter coefficients and a feedback loop. The filter coefficients are reconfigured by a sporadic event (a command from the environment) that activates the sporadic process CoefB.

We see several periodic processes, annotated by their periods, and a sporadic process, annotated by minimal inter-arrival time. We also see inter-process channels – the blackboards and a FIFO, annotated by an arc of the functional priority relation FP. Also the environment input/output channels are shown.

The semantics FPPNs is described in [9] and Appendix A. The main idea is that every pair of processes that share a channel are executed in well-defined relative order determined by (1) their activation times and (2) functional priorities. This relative order should be compatible to the total order derived from zero-execution-time simulation of fixed-priority scheduling. Because the ordering is imposed only between communicating processes, it is a partial order, allowing for parallelism.

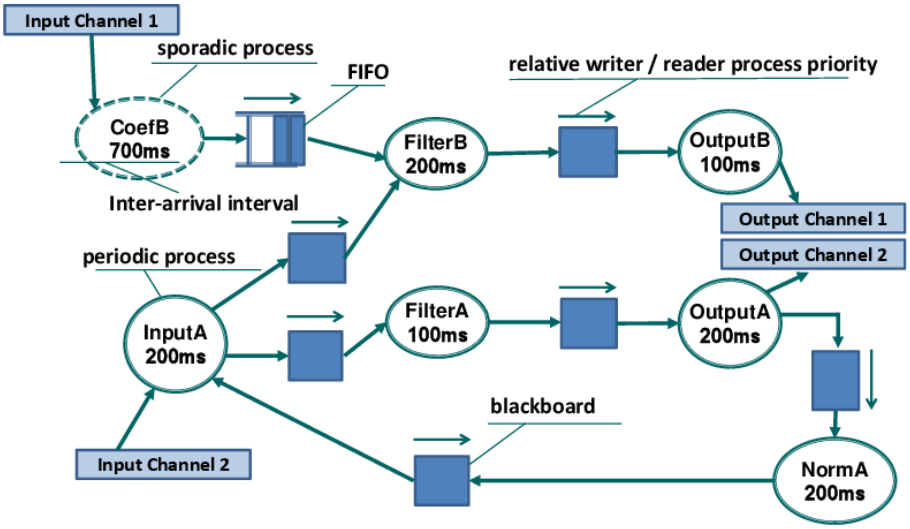

Figure 4: Example of Process Network

4.2 Compiling the processes

The BIP model of a process is automatically extracted from its source code. When translating from a FPPN to a BIP model, the source code is parsed, searching for primitives that are relevant for the interactions between the process and the other components of the system. The relevant primitives are the reads and writes from/to the data channels. Fig. 5 shows the result of compiling the process of Fig. 3 into the backbone language. It can be seen that the behaviour of the resulting automaton is consistent with the behavior of the original source code. The most important difference is how the "reads" and "writes"' are performed. As shown in the figure, each I/O operation is divided into three transitions. Let us consider the read transitions for example. First we have a "Read_Req", which is an external transition which requests access to the channel. After the corresponding interaction the process receives a reference to a memory area where to read from and a validity flag. The next transaction performs the actual read if the validity flag is true. Then transaction "Read_Ack" communicates to the channel that the read has finished. The writing is performed in a similar way.

The FPPN model of computation prescribes certain relative order of execution of communicating processes. The BIP components that impose this order are presented in Appendix B.

4.3 Compiling the Scheduling Policy

Our RT-BIP run-time environment (RTE) currently does not support the interruption of running transitions. Therefore, in our current middleware for time-critical systems we do not yet support preemption. It should be noted that many multi-core platforms choose to not support preemption, instead providing a large number of cores to ensure sufficient degree of multi-threading concurrency without preemption.

We demonstrate programming the scheduling policies in timed automata by considering a policy that combines static-order execution and time-triggering. We base our policy on the policies presented in [10] and partly in [9]. We divide the execution of the FPPN in several slices called "Frames". To each core and each time frame the offline scheduling tool associates a list of jobs that must execute sequentially on the given core in the given time frame. The online schedule is a periodically repeating cycle, where all frames are executed in order: $f_1, f_2, \ldots f_I$, on all cores in parallel, every frame having a fixed duration $T_{f1}, T_{f2}, \ldots$.

In the Gantt chart of Fig. 6 we can see a partial example of a schedule. The BIP implementation of the "Frame 1" of "Core 2" is also shown. The frame components in BIP are specific per frame i and core k. In the beginning, the component synchronizes with other frame components for the same frame i and other cores by interaction $Begin_{fi}$. The jobs are scheduled in a given static order and they all should terminate

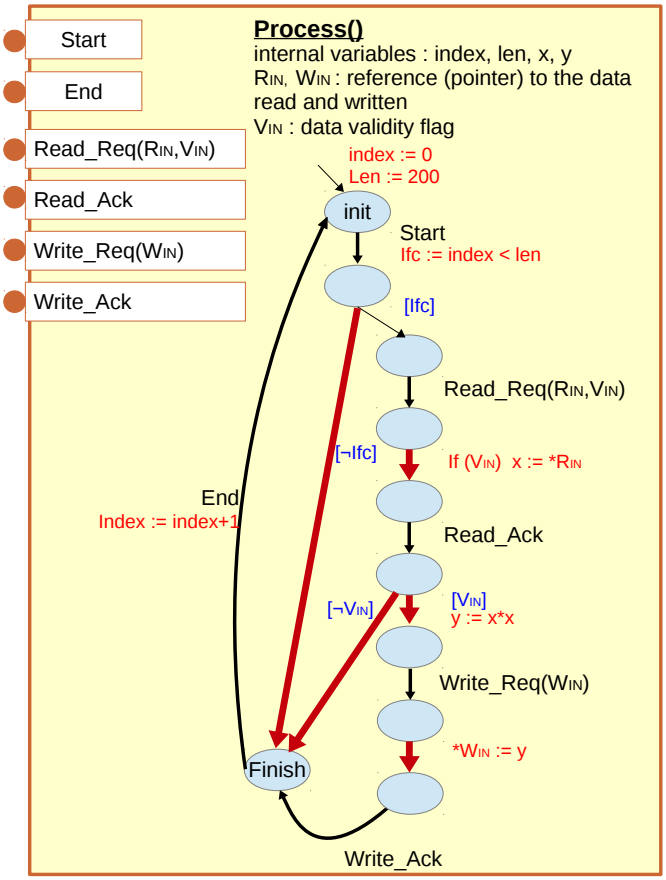

Figure 5: Process translated in BIP

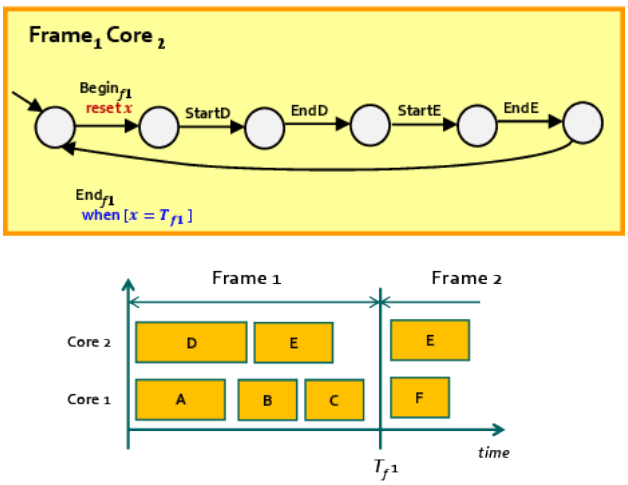

Figure 6: Scheduling Frames

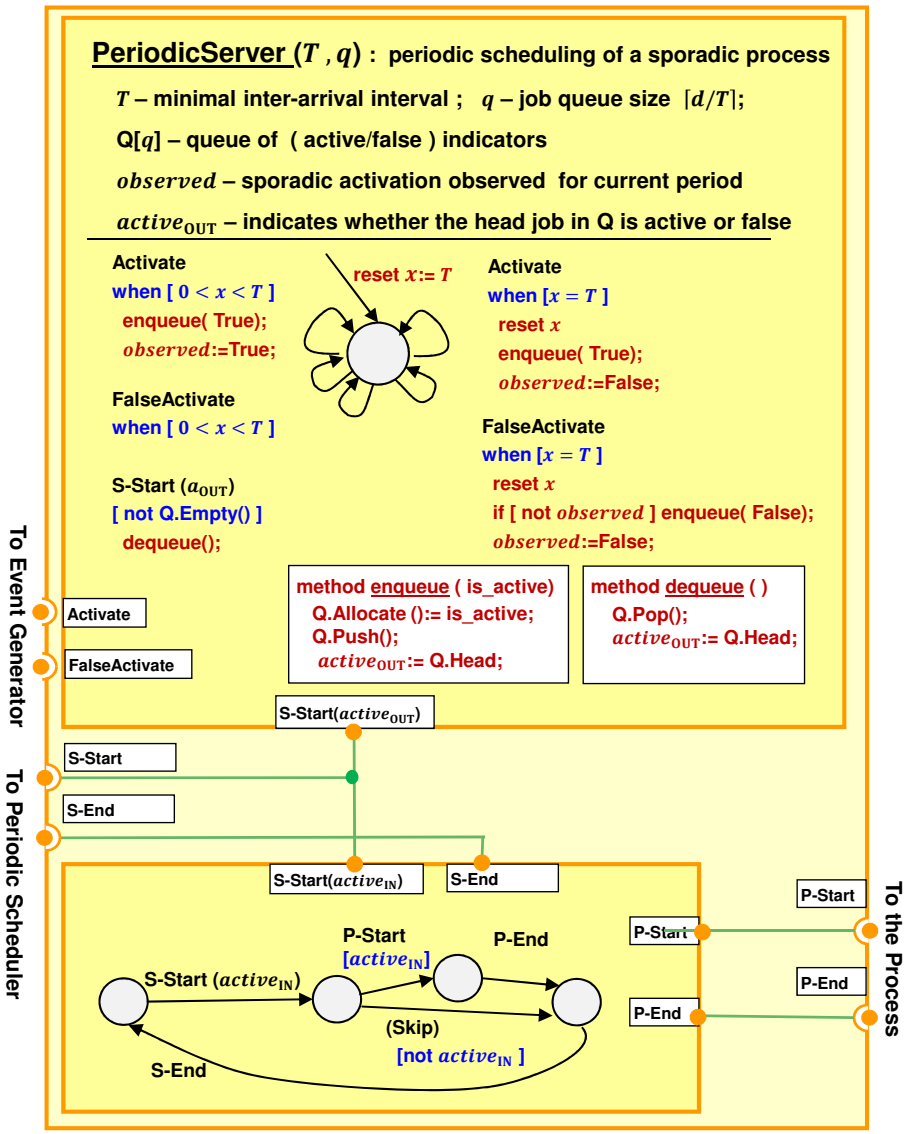

Figure 7: Periodic server, consisting of two sub-components

before End_{fi} interaction, which should occur exactly at time T_{fi}, otherwise the component will deadlock and a run-time error will occur.

In Fig. 7 the periodic server is shown, which is a supplementary adaptor for sporadic processes scheduled in periodic frames. This component manages a queue of active jobs. When a job is activated, it is inserted in the queue, and it is removed when it is scheduled. The queue may contain "false" jobs. This is used for scheduling purposes. We explained above that in each frame we execute jobs in a sequential way. If a frame contains a sporadic job, and this job does not activate, we could have a deadlock in the frame. Thus, to avoid this problem, whenever a sporadic process is not activated, we introduce in the queue a "false job" with zero execution time. The bottom sub-component of the periodic server in Fig. 7 distinguishes between "active" and "false" jobs. In the case of an active job, it signals the job start to the scheduler frame, then to the process, then it waits for the job termination and finally signals it to the frame. In case of a false job, the execution of the process job is skipped. A more detailed explanation of handling sporadic jobs by a periodic server can be found in [9].

Fig. 8 shows how scheduler frames and a sporadic process are connected. The Event Generator generates the activation signal and sends it to the Periodic Server, which triggers the Process in the order defined

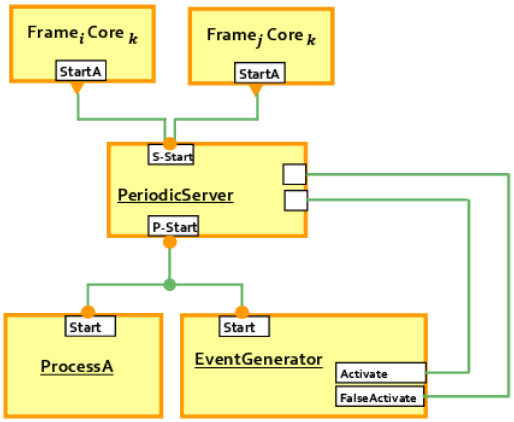

Figure 8: Connection between a Sporadic Process and its Scheduler

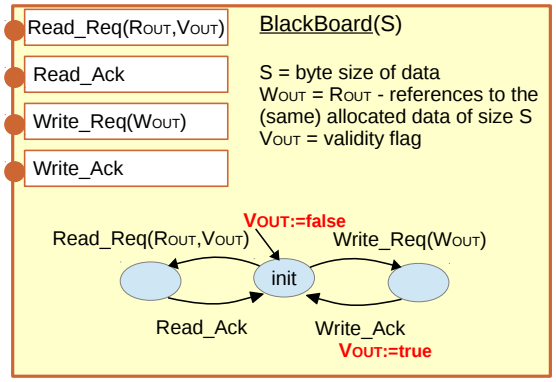

Figure 9: Blackboard

by the frames. For a periodic process, the periodic server is not necessary and the process can be connected directly to its scheduler and generator.

4.4 Compiling the Inter-process Channels

We present here the BIP components that model the data channels, used to "read" and "write" the data communicated between the processes. A basic notion in data channels is the validity flag, introduced earlier. The meaning of this flag is availability of data. In the two types of data channels, the blackboard and the FIFO, it is managed in a slightly different way. The blackboard represents a shared variable, for which it holds by default that once data is written there, it remains available (and hence valid) until it is overwritten by new data.

In the FIFO data items are read and removed in the same order as they are written, and an attempt to read from an empty "queue" or to read more data items than currently waiting in the queue leads to a result whose validity bit is set to "false".

Fig. 9 shows the model for the blackboard. Read(Write) operation is separated into two transactions Read_Req (Write_Req) and Read_Ack(Write_Ack), coherently to the process model shown above. During the request the blackboard communicates to the process the address of the memory where to read from/write to. In case of a read, the validity bit is communicated as well.

The BIP model of a FIFO is shown in Fig. 10. We use the queue data structure introduced in Section 3. The FIFO model is very similar to the blackboard, the main difference is in that here the code for managing the queue is added.

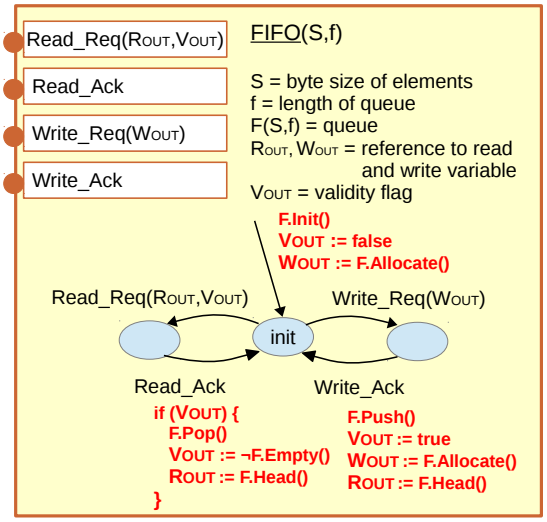

Figure 10: FIFO

4.5 Compiling the Event Generators

We describe here the Event Generator component, individual for each process. The main purpose of this component is to enable the start of jobs after their activation. It also manages the "false" periodic activation for sporadic jobs. The idea of the latter is that for sporadic process p at small intervals $\delta = T_p/K$ for some integer K the environment is polled for the need to activate the sporadic process by calling some platform-dependent subroutine function $protocol()$ that returns a Boolean value indicating activation ('true') or false activation ('false'). The point is that to ensure functional determinism in FPPN MoC, at each moment of time when a sporadic process may potentially get activated it should be always explicitly signaled whether it is activated or not. For periodic processes $\delta = T$ and $protocol()$ always returns 'true'.

The event generator is shown in Fig. 11. It contains a few subcomponents. The Source triggers periodically or sporadically (depending on its protocol) the activation signal. The Sink checks whether any job misses its deadline. For this it uses a "Latency - Burst Shaper" component, which can be seen as a delay line of delay d_p and capacity up to q_p events, where deadline and queue size are process parameters. At activation, the burst shaper starts a new timer (a clock), and when the deadline time has elapsed it enables the output. As shown in Fig. 13, the sink checks whether before the end of the deadline interval at least one job has terminated (pending and not running). It goes into local deadlock state if it is not the case (and this leads to runtime error).

The "Throughput – Burst Shaper" – ensures that the source cannot activate the jobs more than once per time T_p. This subcomponent can be omitted for periodic processes.

The implementation of Source is shown in Fig. 12. This component polls the "protocol()" at periodic intervals δ, as explained earlier. After activation it enables another job to be started by incrementing the counter of active jobs, as a new job can only start if this counter is positive.

Fig. 14 shows how a Burst Shaper is implemented. Its main purpose is to limit the amount of burst to at most σ events per time P where σ and P are given during the definition of the component. This component also signals, via the "Terminate" port, when an activation that arrived P time units ago has elapsed. The main idea of implementation is to use a queue of clock variables implemented as circular buffer.

5 Implementation and Experiments

5.1 Multi-threaded RT-BIP Runtime Environment

As illustrated in Fig. 15, after compiling the application and scheduling into BIP, the BIP design can be partitioned into parts: the schedule components and the application components. The components are

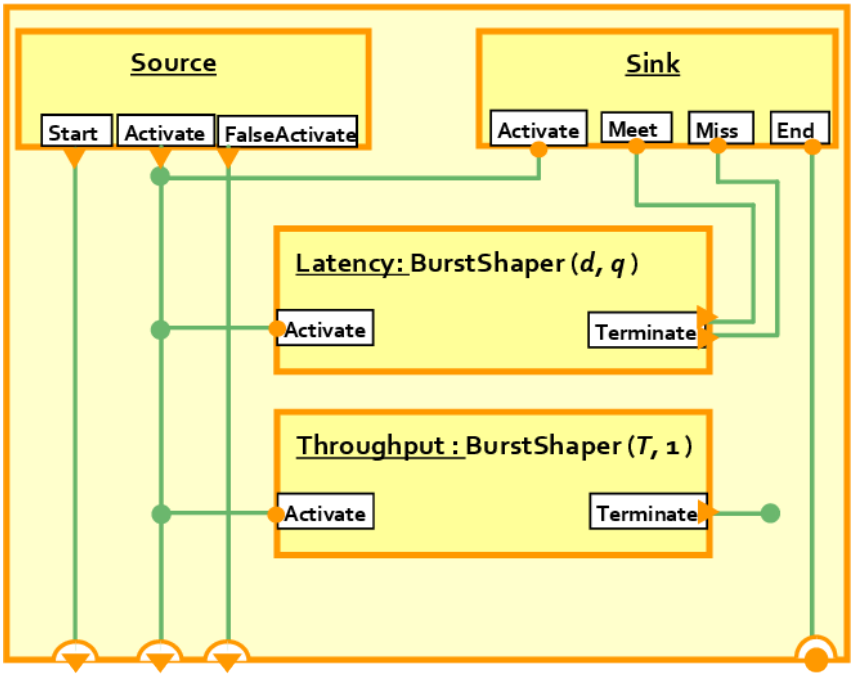

Figure 11: Event Generator

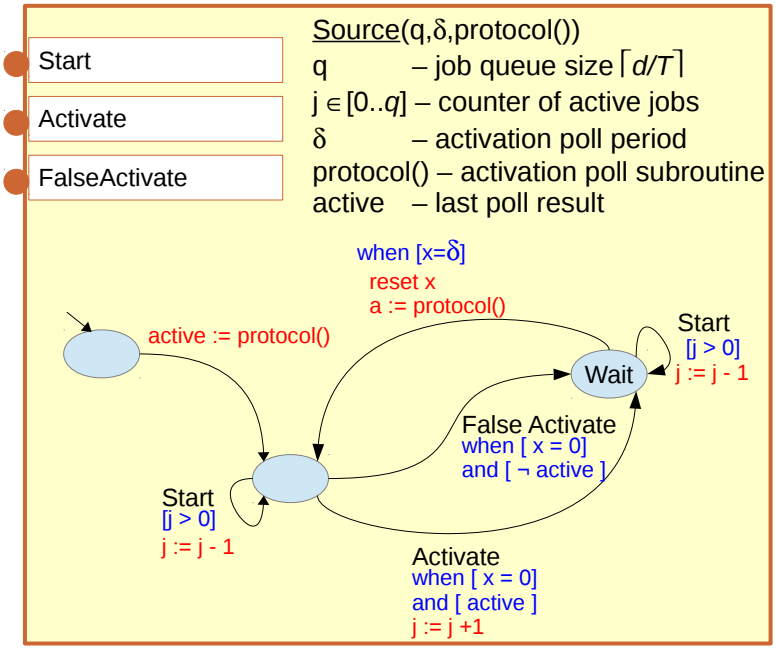

Figure 12: Source

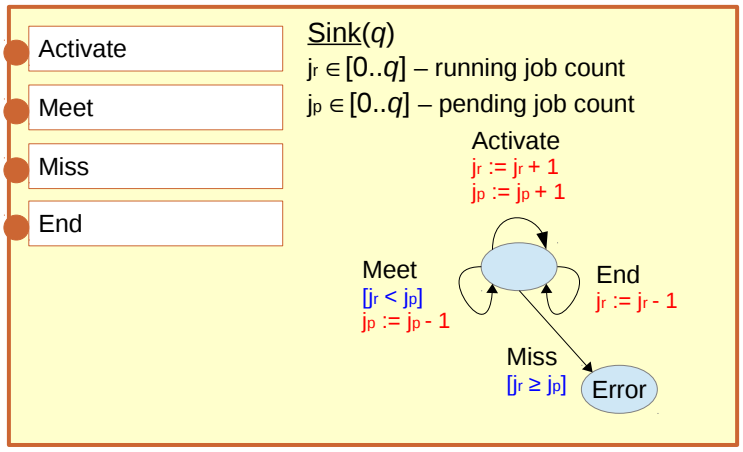

Figure 13: Sink

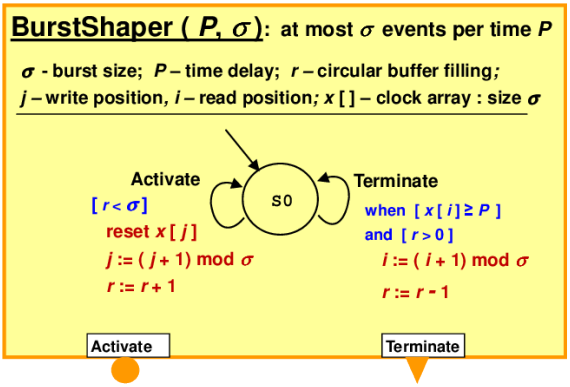

Figure 14: Burst Shaper

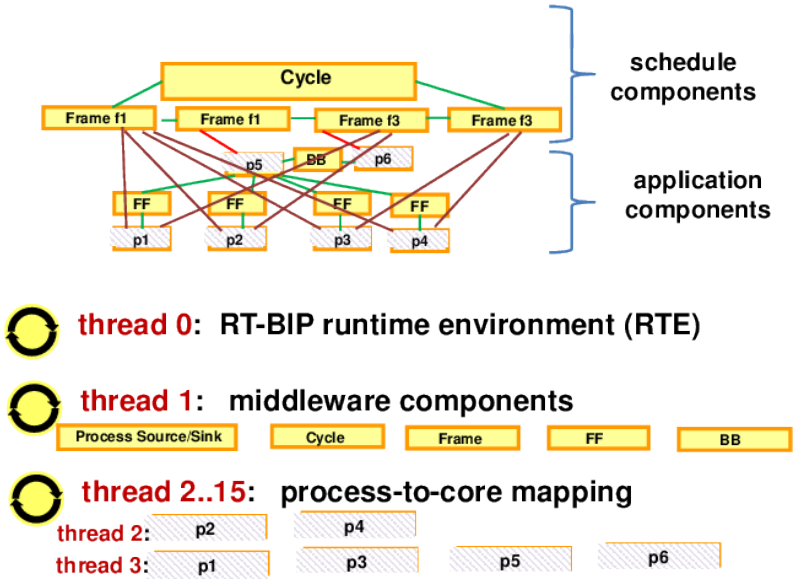

Figure 15: Distributing BIP Components between Cores

joined by BIP connectors, through which they can perform interactions with each other. The application components include the components dedicated to FPPN processes, denoted p1, p2, . . . , and data channels, denoted BB, FF, depending on the type: blackboard and FIFO. The schedule components include one component that models the schedule cycle and a set of components that model frames. The schedule components are connected to the application components to coordinate their execution according to the schedule. The schedule also provides the process-to-core mapping, which is used to generate component-to-thread mapping, illustrated in the bottom part of the figure.

We implemented our framework on Kalray MPPA multi-core architecture inside a single shared-memory cluster. Our framework is based on extension of [14]. The cluster provides 16 processor cores, each one running one POSIX thread. In our framework, Thread 0 executes the RT-BIP run-time environment (RTE), which coordinates the components for the execution according to the RT-BIP semantics. Then, on Thread 1 we run all the middleware components, i.e. all components except the processes. Note that those components can be seen as "instantaneous" components, as they execute only instantaneous transitions. We thus isolate them in a separate thread from the processes, to avoid the risk that continuous transitions of the processes delay the instantaneous transitions of the middleware. Finally, Threads 2..15 are reserved to the process components, which are distributed according to the static schedule computed by an offline scheduling tool.

5.2 Experiments

In this experiment we consider a subsystem of avionics Flight Management System (FMS). Figure 16 shows the application process network. This FMS subsystem is responsible for calculating the best computed position (BCP) and predicting the performance (*e.g.,* fuel usage) of the airplane based on the sensor data and sporadic configuration commands from the pilot, such as configuring the Global Positioning System (GPS). Therefore we have a sporadic process GPSConfig that can execute at most once per 50 ms.

After being pre-processed at "SensorInput", the input data is processed at "HighFreqBCP" process and arrives at "LowFreqBCP" process, which post-processes the data at low frequency and makes it available by other subsystems of FMS. It also provides the results to a feedback loop that takes into account magnetic declination in computing the airplane position. All periodic processes are triggered by event generators and scheduler at period 50ms, but some of them internally skip every k-th execution to execute their true frequency in this multi-rate design. This is not because we do not support different process periods, but to simplify the static schedule.

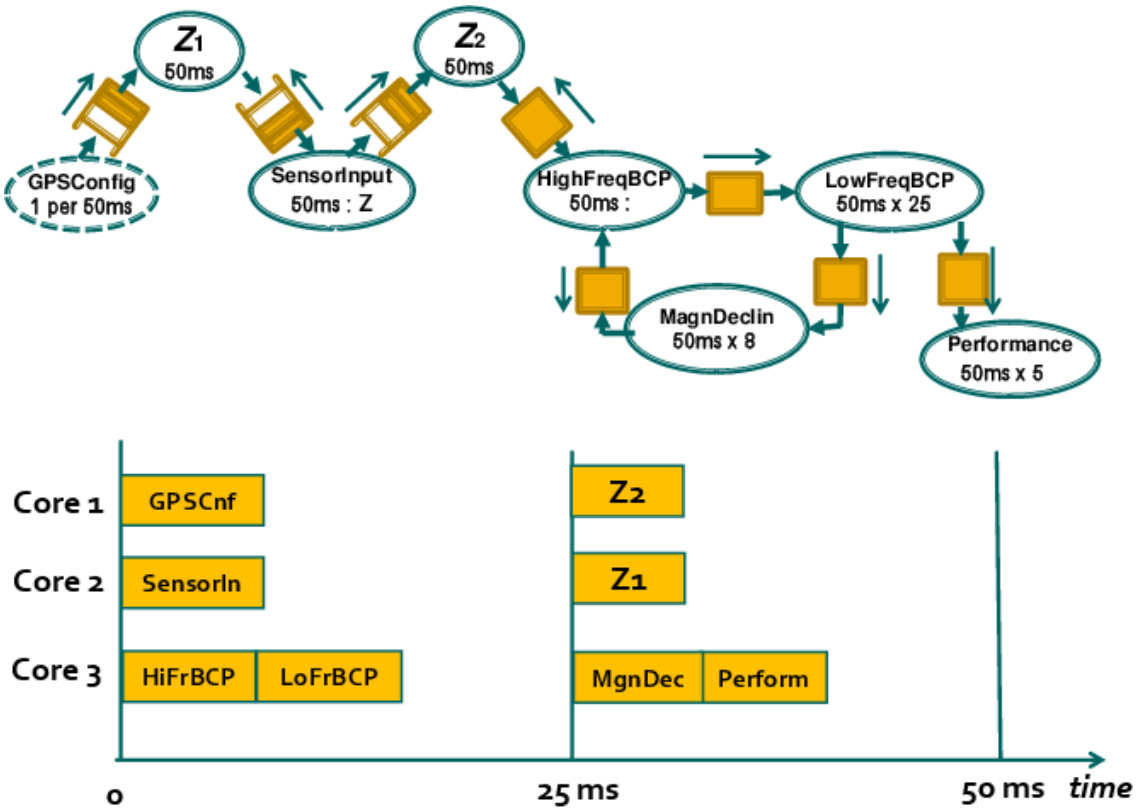

Figure 16: FMS Use Case and its Static Schedule

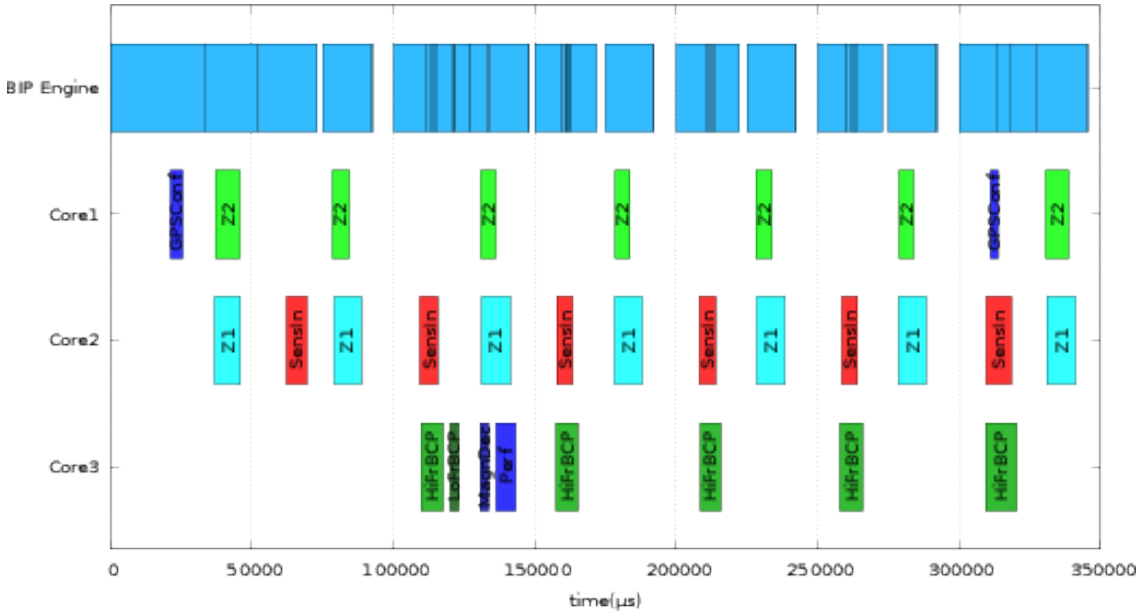

Figure 17: FMS Use Case

To enable pipelining parallelism we use a double buffer approach and insert at some places of the process network processes denoted as Z_k which copy input to the output. Because of this, the buffer is split into two parts and thus the writer can execute at the same time as the reader.

The measured Gantt chart of the execution traces is shown in Figure 17. Studying the chart we conclude that we succeeded in correctly implementing the parallel schedule, but the BIP RTE component synchronization actions turn out to have a large overhead due to inter-core synchronization and cache flushing for memory consistency that occur at BIP interactions. In future work, we see a large room for improvement to reduce this overhead. Note that this overhead is relatively large compared to process execution because of fine granularity of the process computations, for more coarse-grained processes the relative overhead of BIP interactions would be smaller.

6 Conclusions

In this paper we proposed a common approach to program not only application functionality, but also the middleware for time-critical systems. We proposed to use for this purpose combined timed-automata/procedural languages that support deployment on multi-cores in multiple threads. We demonstrated this approach on a concrete model of computation and scheduling policy, and implemented it in a publicly available tool [5]. To confirm the validity of our approach we showed the results for deploying a real-life avionic use-case on a real multi-core platform.

The proposed approach potentially opens the possibility of unifying different MoCs and scheduling policies in common frameworks. In future we will work on automatic support of different MoCs and scheduling policies, including preemptive ones, such as EDF. Middleware for real-time application on multicore.

References

[1] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "A timed-automata based middleware for time-critical multicore applications," in *SEUS*, 2015. 1

[2] E. A. Lee, "Absolutely positively on time: what would it take?," *Computer*, vol. 38, no. 7, pp. 85–87, 2005. 1

[3] H. Fuhrmann, J. Koch, J. Rennhack, and R. von Hanxleden, "Model-based system design of time-triggered architectures—an avionics case study," in *25th Digital Avionics Systems Conference (DASC'06)*, (Portland, OR, USA), October 2006. 1

[4] R. Le Moigne, O. Pasquier, and J.-P. Calvez, "A generic rtos model for real-time systems simulation with systemc," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3*, DATE '04, (Washington, DC, USA), pp. 30082–, IEEE Computer Society, 2004. 1

[5] P. Poplavko, P. Bourgos, D. Socci, S. Bensalem, and M. Bozga, "Multicore code generation for time-critical applications, http://www-verimag.imag.fr/multicore-time-critical-code,470.html." 1, 2, 6

[6] T. Abdellatif, J. Combaz, and J. Sifakis, "Model-based implementation of real-time applications," in *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, ACM, 2010. 1, 2, 2, 3

[7] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times a tool for modelling and implementation of embedded systems," in *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pp. 460–464, Springer, 2002. 2

[8] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, "The IF toolset," in *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, pp. 237–267, 2004. 2

[9] P. Poplavko, D. Socci, S. Paraskevas Bourgos, and M. B. Bensalem, "Models for deterministic execution of real-time multiprocessor applications," in *DATE'15*, 2015. 2, 4.1, 4.1, 4.3, 4.3, B

[10] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pp. 1–15, IEEE, 2013. 2, 4.3, B

[11] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, "Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset," in *19th International Conference on Real-Time and Network Systems*, 2011. 2

[12] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "Compsoc: A template for composable and predictable multi-processor system on chips," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, p. 2, 2009. 2

[13] R. Alur and D. L. Dill, "Automata For Modeling Real-Time Systems," in *Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)* (M. Paterson, ed.), vol. 443 of *LNCS*, pp. 322–335, Springer, 1990. 3

[14] A. Triki, J. Combaz, S. Bensalem, and J. Sifakis, "Model-based implementation of parallel real-time systems," in *FASE'13*, Springer, 2013. 5.1

Appendices

A FPPN Semantics

The *semantics* of the process networks assume that all simultaneous process activations are signaled synchronously, *i.e.,* all together at the same event. Based on this assumption, there are two variants to define the semantics: *zero-delay* and *real-time*.

Zero-delay semantics defines how to simulate the process networks for functional simulation. It assumes that the process jobs are executed sequentially. It requires that in the sequence of jobs produced by simulation the following '**Rules**' apply:

Rule (1) Every job occurs immediately after the activation and taking a zero time, therefore the *jobs are executed in the order of their activations*

Rule (2) If jobs of processes p_1 and p_2 are activated simultaneously:

- if $(p_1, p_2) \in FP$ then p_1 executes its job earlier than p_2
- if $(p_2, p_1) \in FP$ then p_2 executes its job earlier than p_1
- in the remaining case the relative order of jobs is arbitrary

Rule (1) and Rule (2) are equally important for *functional determinism*. They ensure that the order of reads and writes to any blackboard and mailbox is function of the time stamps of the activation.

While the above rules define zero-delay functional simulation, the *real-time semantics* defines how the process networks can be executed on the real-time platforms, where every action takes some physical time and some actions can be executed in parallel. Within some limits, we allow the jobs to have any execution time and to start concurrently with each other at any time after their invocation. However, the jobs should satisfy *timeliness constraints* and *precedence constraints*. Timeliness means completion within the deadline after the activation. The **precedence constraints** apply for two '**Cases**':

Case (I) Subsequent jobs of the same process

Case (II) Jobs of process pairs connected by mailbox or blackboard

The precedence constraints dictate that in both Case (I) and Case (II) we should expect the ordering defined by Rule (1) and Rule (2) above, *i.e.,* the job jobs should occur in the order of activation and when two functionally-related processes are activated synchronously then they should be fired at the order defined by their relative priority. Therefore, for (I), we *forbid auto-concurrency*[3] and for (II) we *forbid data races* on the channels. This prevents non-deterministic updates of the local state of the process or of the data shared between the processes.

B Compiling FPPN Functional-Priority into BIP

For functional determinism purposes, the FPPN semantics imposes precedence constraints on the order of execution. When FPPN network is compiled into BIP, for the Case (I) defined in the previous section the precedence constraints are satisfied by construction, because BIP components for processes never start a new job execution until the previous one for the same process has finished.

Case (II), however, requires that any pair of communicating processes follow Rule (1) and Rule (2), in particular that the process that 'competes' with another process (being activated simultaneously), but has higher functional priority should be executed earlier. Figure 18 shows a BIP component "Functional Priority" that ensures satisfaction of precedence-constraint Rules (1) and (2) for a given pair of processes "A" and "B", assuming $(\mathrm{A}, \mathrm{B}) \in FP$, *i.e.,* that "A" has a higher functional priority. For each pair of communicating processes one such component should be inserted and plugged into the respective connectors.

[3] auto-concurrency requirement could be elevated for processes that do not have a local state

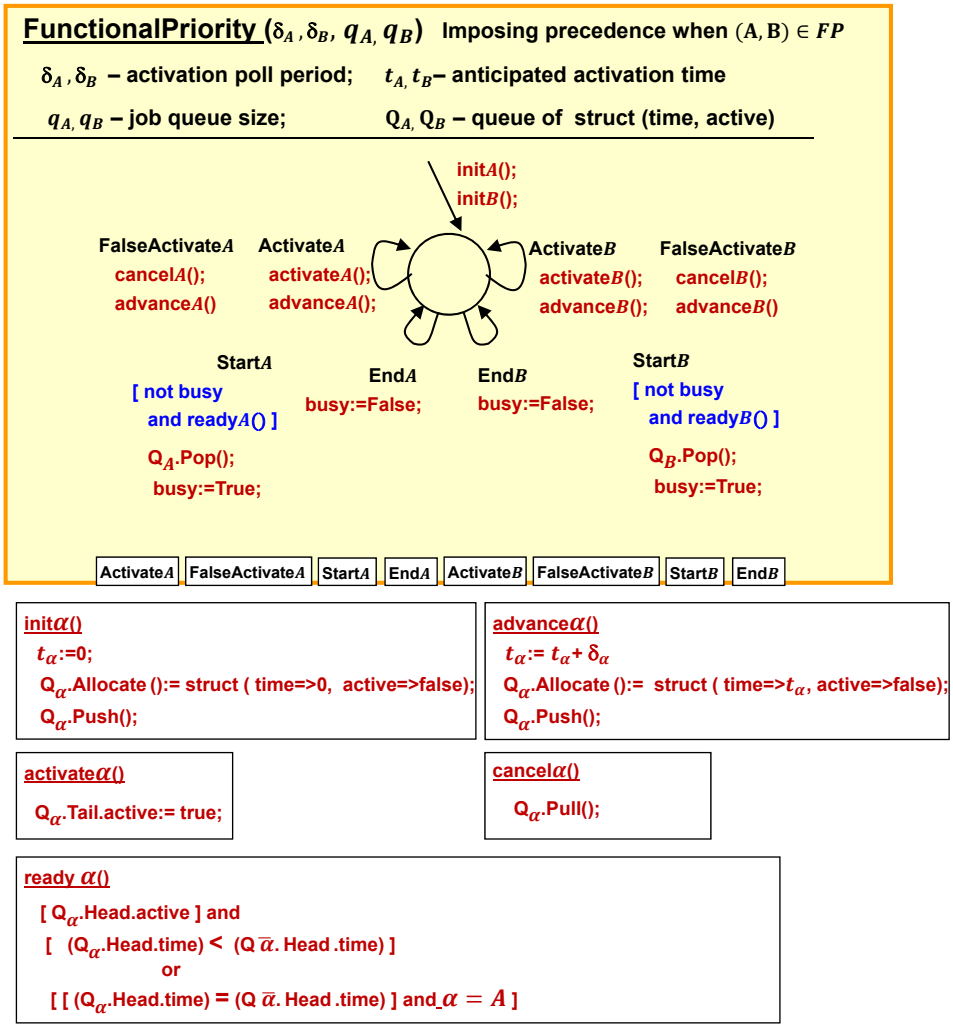

Figure 18: Imposing precedence order of executing “A” and “B” where “A” has higher functional priority

Recall that the evolution of every job execution goes through three steps: 'activated', 'started' and 'ended'. The "Functional Priority" component handles the three steps of both processes in almost symmetrical way, except that there is some asymmetry in the method that determines whether the job is ready: if two jobs are simultaneously activated then first the job of process "A" gets ready in line with Rule (2) and then, after it has executed, job of process "B" will become ready.

The "Functional Priority" component keeps two job queues denoted Q_α where $\alpha \in \{A, B\}$ indicates a process selection. In our notations we assume that $\overline{\alpha}$ means 'other than α', *i.e.,* if $\alpha = A$ then $\overline{\alpha} = B$ and if $\alpha = B$ then $\overline{\alpha} = A$.

The "Functional Priority" component receives either 'Activateα' or 'FalseActivateα' from the event generator of process 'α' at regular time intervals with period δ_α. Based on this, the component updates the status of the next process job at the tail of the job queue. In the case of the 'false' activation (*i.e.,* indication that no job is activated), the job is pulled away from the tail. Note that the queue tail always contains information about the next anticipated activated job thanks to methods 'initα' and 'advanceα', but this anticipated job is conservatively marked to be not active until the corresponding 'Activeα' transition.

Note that the scheduling-policy should satisfy the precedence constraints, and the "Functional Priority" components can be seen as part of scheduling-policy components. "Functional priority" components are not necessary if the scheduling-policy components themselves already impose that the components execute in precedence-constrained order. For example, separation of jobs into frames and giving them sequential order within the frame in TTS scheduling policy of [10] already impose precedence constraints, when compiling systems scheduled by TTS we disable insertion of "Functional Priority" components as they are not necessary. On the other hand, the cyclic frames that implement the static-order policy of [9] rely on "Functional Priority" components to ensure proper execution order of processes mapped to different cores.

Note that one may also use "Functional Priority" components as the only components that model scheduling policy. In this case we have so-called ASAP (as soon as possible) policy where every process executes as soon as it has arrived and its predecessors have finished. This policy may require multiple cores to be available at the same time to execute multiple independent processes concurrently.