**International Journal on Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# Correct-by-construction Model-based Design of Reactive Streaming Software for Multi-core Embedded Systems

**Fotios Gioulekas · Peter Poplavko · Panagiotis Katsaros · Saddek Bensalem · Pedro Palomo**

Received: date / Accepted: date

**Abstract** We present a model-based design approach towards correct-by-construction implementations of reactive streaming software for multi-core systems. A system's implementation is derived from a high-level process network model by applying semantics-preserving model transformations. The so-called Fixed Priority Process Networks (FPPNs) are programmed independently from the execution platform and combine streaming and reactive control behavior with task parallelism for utilizing multi-core processing. We first define the FPPN sequential execution semantics that specifies precedence constraints between job executions of different tasks. Applications are thus rendered such that, for any given test stimuli, a deterministic output response is expected. Furthermore, we define the FPPN real-time semantics based on a timed-automata modeling framework. This is provably a functionally equivalent semantics specifying the real-time execution of FPPNs and enabling runtime managers for scheduling jobs on multi-cores. A model transformation framework has been developed for deriving executable implementations of FPPNs on the BIP (Behavior - Interaction - Priority) runtime environment, ported on multi-core platforms. Schedulability is established by static analysis of the FPPN and it is guaranteed by construction. Thus, the developers do not need to program low-level real-time OS services (e.g. for task management) and applications are amenable to testing, as opposed to if their outputs would depend on timing behavior. We have successfully ported a guidance-navigation and control application of a satellite system, onto a radiation hardened multi-core platform. Various implementation scenarios for efficiently utilizing HW resources are illustrated and the test results are discussed.

**Keywords** Process network · Model of computation · Model transformation · Timed-automata · Critical systems · Multi-core processors

This work was partially supported by the European Space Agency project MoSaTT-CMP under Contract No. 4000111814/14/NL/MH

F. Gioulekas
Department of Informatics, Aristotle University of Thessaloniki, Greece
E-mail: gioulekas@csd.auth.gr

P. Poplavko
Mentor®. A Siemens Business. Montbonnot, France
E-mail: petro.poplavko@siemens.com

P. Katsaros
Department of Informatics, Aristotle University of Thessaloniki, Greece
E-mail: katsaros@csd.auth.gr

S. Bensalem
Université Grenoble Alpes (UGA), VERIMAG, Grenoble, France
E-mail: Saddek.Bensalem@univ-grenoble-alpes.fr

P. Palomo
Deimos-Space®, Madrid, Spain
E-mail: pedro.palomo@deimos-space.com

## 1 Introduction

The advent of multi-core processors, and their adoption into modern embedded systems, has enhanced the possibilities to integrate more functions into a single CPU chip while sustaining power-consumption in low-levels. However, the design of timing-critical software still faces important challenges [39], since in contrast to hardware design languages which inherently support timing and parallelism, imperative programming languages do not integrate appropriate features for these two aspects. To this end, the model-driven approach in software design based on Models of Computation (MoC) for real-time systems is an attractive alternative.

For streaming signal processing software, the synchronous data-flow languages [33] provide support for task-level concurrency, suitable for exploiting multi-core parallelism [30]. The programs take the form of directed graphs with nodes representing their functions and arrows for the data flows between them. The programs' timing behavior is rendered predictable through statically scheduling their functions [32]. However, this programming paradigm is not appropriate for timing-critical applications that exhibit reactive behavior, such as flight control software, which are expected to react to stimuli from the environment within strict time bounds. In programs written in reactive-control synchronous languages (*e.g.,* Esterel, Lustre) [25], computations are structured in sequences of logical clock ticks, thus eliminating the non-determinism from the interleaving of concurrent behaviors. Such programs are amenable to formal verification and executable code generation, but these languages do not support task parallelism and scheduling with timing constraints on multiprocessors.

It is therefore evident that selecting a single MoC for the model-based design of applications that combine streaming and reactive control processing is challenging, which is aggravated by the fact that the popular MoCs and the widely used real-time scheduling policies [8, 18] are hardly integrated. The problem is even harder in multi-core systems, where scheduling policies may be more heterogeneous following the need to address various sources of resource interference [36, 39]. To this end, we presented in [22] the semantics of the *Fixed Priority Process Network* (FPPN), a MoC where task invocations depend on a combination of periodic data availability (similar to streaming models) and sporadic control events. Moreover, in [38], we proposed static scheduling methods for FPPNs that demonstrate a predictable timing behavior on multi-cores. With the FPPN MoC we aim to address the following concerns in model-based design of reactive streaming software for multi-cores:

- ensure that the application's outputs depend only on the event generation time stamps and the input data sequences (functional determinism), i.e. they do not depend on the timing behavior of the implementation;
- developers should not be obliged to program the application using low-level real-time OS services (e.g. for task management, inter-task communication, memory allocation etc.) and they only need to reason in terms of high-level schedulability concepts (e.g. tasks, priorities, deadlines, offsets etc.);
- derive a correct by construction implementation through semantics-preserving model transformations used to generate code for FPPNs on the multi-core platform.

The latter concern is important for ensuring the schedulability guarantees of the statically analyzable high-level design. In [22], we presented a comprehensive FPPN semantics definition. First, the FPPN sequential execution semantics was defined, which introduces precedence constraints between job executions of different tasks. In current article, we prove that this semantics ensures the first mentioned concern, i.e. that of functional determinism. At a lower level, we have introduced in [22] the FPPN real-time semantics, which relaxes the sequential order of execution and the zero-delay assumptions, thus allowing for task parallelism. In the current article, we show that the latter semantics is functionally equivalent to the former and thus both semantics describe similar functionally deterministic behaviors of the FPPN.

To ensure correctness-by-construction (last concern), a model transformation framework has been implemented that enables programming FPPNs at a high level through integrating them into an architecture description interface. The present article provides a detailed account of this framework and the supported software design flow. Based on the FPPN MoC, our framework[1] provides means for gradual refinement of real-time multitasking software design. With respect to this, we show how to explore various design scenarios - in terms of task interactions and scheduling - for ensuring timeliness, and how the derived system's implementation is executed on multi-core CPUs. The proposed methodology is evaluated and demonstrated by porting a real satellite on-board application onto the European Space Agency's quad-core Next Generation Microprocessor (NGMP) platform [2].

In summary, this article extends the FPPN semantics definition in [22] with additional contributions towards a model-based design framework for addressing the three aforementioned concerns, as follows:

- We restate the FPPN sequential execution semantics from [22], also called zero-delay semantics, and we prove here that the state-variables of the FPPN entities are not updated non-deterministically.
- We also restate the FPPN real-time semantics defined using an executable formal specification language called BIP (Behavior - Interaction - Priority) [4], for modeling networks of connected timed automata components.
- The fundamental FPPN correctness properties are formulated using Linear Temporal Logic (LTL) [6]. We show that both semantics fulfill these properties, thus concluding that the functional determinism is

[1] The framework is online at [3]

also satisfied by the real-time semantics and that the two semantics definitions are functionally equivalent.
- We present our amendments to the TASTE toolset [35] (first shown in [23]), which allow embedding functional code into an FPPN model through a high-level architecture description interface.
- We present the set of code-to-code and graph-to-graph rewriting rules that define the FPPN to BIP (FPPN2BIP) model transformation founded on the functional equivalence between the two semantics definitions. The FPPN2BIP model transformation is instrumental for the overall model-based design flow that is presented in this article.
- The FPPN scheduling tools [38, 40] accept as input a task graph, which dictates job executions and is used to determine the mapping of FPPN entities to processor cores. We adapt the task graph generation algorithm from [38], for deriving the graph directly from the TASTE FPPN model.
- We provide experimental results from various implementation scenarios of a guidance - navigation and control application on ESA's quad-core NGMP platform (one implementation scenario was shown in [22]). Through these implementation scenarios we show how one can intervene in various stages of our design flow, for improving the resource utilization, without violating the FPPN semantics.

The rest of the paper is organized as follows. Section 2 reviews the related work in comparison with our model-based framework. Section 3 gives an overview of our design approach, while characterizing and motivating its key ingredients. Section 4 introduces the necessary definitions in regard to the FPPN MoC. Section 5 refers to the FPPN zero-delay semantics, while Section 6 describes the FPPN real-time semantics. In Section 7 we introduce the FPPN correctness properties and we show that the two semantics definitions are functionally equivalent. Section 8 presents the TASTE representation of FPPN models, our FPPN2BIP model transformation and the overall model-based design flow. Section 9 exposes our experimental results from porting the guidance, navigation and control application on ESA's NGMP platform. Finally, Section 10 concludes the paper and gives insights on future research directions.

## 2 Related work

In the related bibliography, there is a number of relevant MoCs and model-based design flows, but no other approach for reactive streaming software on multi-cores combines characteristics that address all three main concerns mentioned in Introduction.

In general, existing model-based design flows are based on an architecture description, which enables transformations for deriving models used to analyze the system's non-functional properties with appropriate tools [29]. The AADL (Architecture Analysis and Design Language) language is often used for the modeling and analysis of real-time applications. In [43, 44], the authors propose a formal semantics for AADL using Timed Abstract State Machines, whereas in [34] the authors introduce model transformations to the LNT language, in an attempt to support formal system verification. Schedulability depends on necessary assumptions for the temporal and concurrency properties of computations, which render a model statically analyzable. Applications are therefore dependent on the execution platform, which has to fulfill certain assumptions.

The TIMES tool [5] supports the design of timing-critical software using timed automata and provides code generation functionality, but it is not appropriate for the design of task-parallel software through a high-level MoC. In [41], the authors present a design approach based on parallel timed automata, modeled in BIP, that supports the generation of task-parallel code. However, that work does not introduce model transformations, which allow deriving a system's implementation from a high-level MoC. The use of a MoC allows decoupling the model specification from the system's implementation.

Kahn Process Networks [31] (KPNs) are perhaps the most important related MoC. Their main features are functional determinism and inherent support of parallel and distributed implementations. These features render KPNs very popular for research on parallel and distributed embedded systems based on multiprocessors. On the other hand, KPNs do not support the notion of time and reactive behavior, and in general case they are not schedulable. The FPPN MoC differs from KPN in several ways. To support reactive behavior and scheduling we let the processes execute in steps (jobs). The steps are activated by potentially aperiodic events whose relative timing is important for the functional behavior, as opposed to KPN where the relative timing of data arrival in different channels does not change the function. Also, in a FPPN the access to the data-channels is non-blocking, whereas in KPNs it is blocking. Nevertheless, similarly to KPNs, the FPPNs ensure functional determinism, while following the same intuitive and popular parallel-programming paradigm of a set of parallel processes communicating via channels.

In model-based environments like Ptolemy II [15] and PeaCE [24], a design model is first built, which enables a refinement and prototype evolving procedure through incorporating diverse MoCs. Schedulability aspects are often ignored in widely used MoCs. The data-flow MoCs

that stem from the KPNs have been extended to support the timing constraints of signal processing applications and to offer the respective scheduling algorithms. There are also advanced embedded system design frameworks like CompSoC [26] based on these MoCs, with an impressive set of application benchmarks. These models inherit from KPNs their functional determinism and the natural support of distributed programming, but unfortunately they also inherit the lack of support of reactive behavior. In the Prelude toolset [14], the user specifies multi-rate synchronous systems using a synchronous reactive MoC (strictly single-rate periodic systems). However, due to the MoC's expressive power, it is hard to derive schedulability analyses, unless restricting its semantics. The reactive process networks (RPN) [19] is a MoC that does not support scheduling with timing constraints; however, RPNs provide the fundamental semantics definition for consolidating the streaming and reactive control behaviors. In particular, they have introduced an important principle of performing *maximal execution run* of a data-flow network in response to a sporadic control event, which ensures that these events have a predictable effect of the evolution of system state. In the semantics of FPPN presented here, we reuse this principle which helps us to ensure the functional determinism of our MoC.

The Distributed Operation Layer – Critical (DOL-C) [21] presents a MoC where streaming and reactive behavior are combined and the design framework supports the deployment of scheduled real-time applications for multi-cores. Our work in this article is partly an important complement to [21] and partly a significantly different new work. Unlike DOL-C, here we present a formal definition of a zero-delay semantics for the FPPN, which ensures functional determinism by construction. On the other hand, the functional behavior of a poorly designed DOL-C model may depend on scheduling and on the execution platform. This is prevented in our framework, since the model transformations preserve the functional behavior by construction. Finally, unlike the DOL-C to BIP transformation, our design framework is integrated with TASTE, a professional open-source environment for software design based on formal modelling and formal validation.

Similar to DOL-C, another timing-aware reactive MoC that does not guarantee functional determinism but provides real-time scheduling is the DMPL (DART Modelling and Programming Language) [12]. This framework uses advanced real-time scheduling algorithms and has very extensive support for advanced distributed applications. In contrast to DMPL, FPPN fits best for concurrent but not for distributed systems, due to the assumption of synchronous invocation of the tasks, which is not straightforward to implement in distributed environments. Nevertheless, FPPN fits well to implement an individual distributed-system node, open for communication with other nodes. We note that we are not aware of any framework for distributed systems that supports reactive control behavior and functional determinism at the same time.

The model-based environment AutoFOCUS [28] and its derivatives AutoFOCUS2 [10], and AutoFOCUS3 [1, 27] support a seamless development process for the specification and the design of distributed and concurrent reactive control systems based on rigorously defined formal semantics. The AutoFOCUS tool is based on the FOCUS framework [9, 11], which is a formal development method for specifying a system, its interface and behavior, while supporting component decomposition and refinement at various abstraction layers. Similar to TASTE that we use to implement FPPNs, AutoFOCUS enables a structured and refinement-based approach to formally specify concurrent systems, but its support to real-time scheduling concepts is still under exploration.

In Table 1 we summarize the main features of the mentioned MoCs. The 'yes' indication means that the referred feature is either claimed by the authors or seems straightforward to attribute to the given MoC. The 'no' indication means the opposite, which does not necessarily imply impossible to support.

Table 1: The MoC features claimed the literature

| MoC | Schedulable | Reactive | Distributed | Func. determ. |
|---|---|---|---|---|
| Data-flow (CompSoC) | yes | no | yes | yes |
| KPN | no | no | yes | yes |
| RPN | no | yes | no | no |
| DOL-C | yes | yes | yes | no |
| DMPL | yes | yes | yes | no |
| FPPN | yes | yes | no | yes |

From Table 1 we see that FPPN offers a unique combination of important process-network features. Note that though we claimed 'no' support of 'functional determinism' in RPN, it seems possible to prove it if adopting certain extensions, *e.g.,* priorities of events, as mentioned by the authors themselves.

## 3 The FPPN-based Design Flow

This section provides an overview of the design flow based on the FPPN model, in order to place the overall approach within the wider area of real-time concurrent system design. While the technical details are postponed until Section 8, after the definition of the FPPN semantics, here we focus on: (1) what exactly is offered by

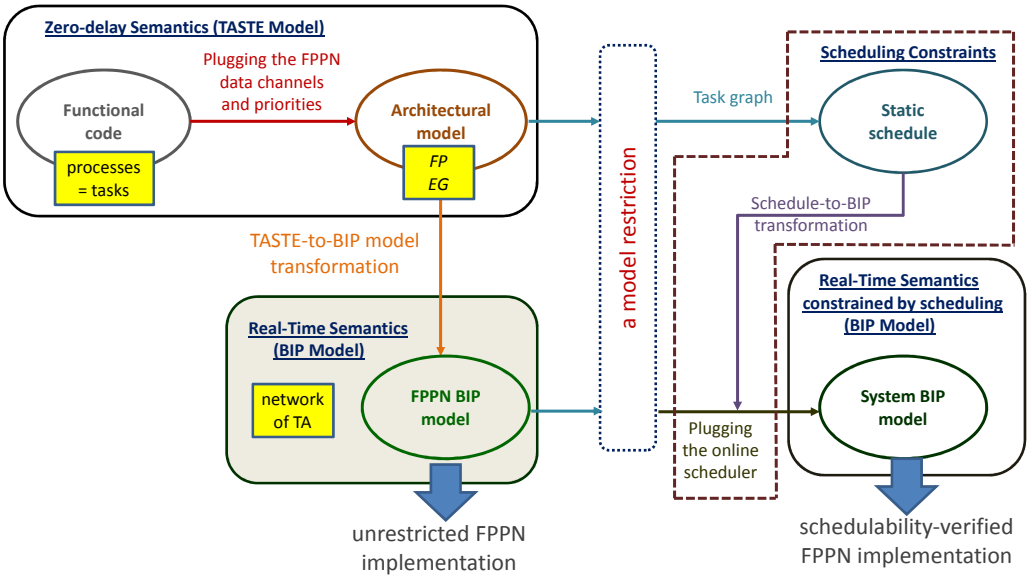


Fig. 1: Design flow based on FPPN application specification

*FP* - functional priority relation between tasks – precedence order

*EG* - event generators – (a)periodic events invoke tasks synchronously

TA - time automata, in BIP (Behavior Interactions Priorities) language, executable on top of BIP RTE

the FPPN-based design flow, (2) what exactly are the model's distinct features and (3) what kind of guarantees does it provide. Figure 1 illustrates our design flow starting from the specification and ending in system implementation.

First, the user provides the functional code of the processes (also called tasks) in an imperative language and connects them with each other within a high-level platform-independent architecture model using the structural primitives of the FPPN MoC. At the specification stage the user does not have to care about the task delays, the number of processors and for how to fulfill the task deadlines, but he focuses only on the correct and reproducible functional behavior.

To ensure this, we define the so-called zero-delay (ZD) semantics for FPPNs, which adopts for simplicity the assumption that tasks are executed instantaneously and sequentially. Such a semantics definition avoids all unnecessary implementation-related details for task synchronization and is easier to be formally specified The input and output data of an FPPN network are received and sent as samples labeled by timestamps based on their physical timing. The ZD semantics guarantees the *functional determinism*, *i.e.,* that the contents of output data samples is a well-defined function of the input data samples and their timestamps. The value of guaranteeing functional determinism through this definition is grounded to at least two arguments. First, for correctly implementing the control laws in reactive control systems, the data calculated by a control law should depend functionally on the data and timing of inputs; if there is any jitter in timing or non-determinism in data, then the quality of control may be impaired in unpredictable way. Secondly, when testing and debugging a functionally deterministic system, a bug can be always reproduced by applying the same input sequence as the one applied when the bug was manifested.

An important ingredient of an FPPN architecture definition is the specification of the functional priority relation $\mathcal{FP}$, which is defined by an acyclic graph with the tasks as nodes and represents a relative order of task executions, whenever the tasks are invoked simultaneously. This relation has to be established between tasks that exchange data between each other. The basic idea for specifying $\mathcal{FP}$ is to integrate a graph showing the sequence order of data-communicating tasks as intermediate computation stages in a control flow. The ZD semantics can be simulated by executing a classical fixed priority schedule on a single processor using a priority assignment consistent with $\mathcal{FP}$. The advantage of this approach is its support for aperiodic (sporadic) tasks. Fixed priority scheduling adapts the ordering of tasks dynamically in a way that is functionally dependent on the (dynamic) invocation time of such tasks. Therefore, functional determinism can be ensured even without imposing synchronization constraints between

the tasks [17]. However, though functional determinism is not a problem for uni-processor systems this property of fixed priority scheduling is lost when instead of a single-core processor multiple cores are used. This poses a challenge in designing multi-core real-time systems [13]. Therefore, to address this challenge we simulate the single-core fixed priority in the ZD semantics and then we ensure an equivalent behavior on multiple cores by imposing the respective synchronization constraints. This is achieved in the so-called real-time (RT) semantics, which is discussed further below.

Another ingredient of an FPPN architecture definition (Figure 1) are the event generators $EG$. There is one event generator per task and they work altogether in tandem with $\mathcal{FP}$ to ensure the functional determinism. Informally, an event generator for a given task specifies a 'law' respected by the timestamps of subsequent invocations. This law can be periodic, sporadic, or even user-defined, since in our framework the user can define his own 'timestamp appearance law' using a timed automaton model. The timestamps of an event generator are imposed not only on a given task, but also on the subset of external input/output channels accessed by the task. We assume synchronicity of $EG$, *i.e.,* for each moment of time it is known which tasks have been invoked synchronously (simultaneously). In ZD semantics, the processes access their external channels and exchange data at the moment of their invocation, while following an execution order consistent with $\mathcal{FP}$. This ensures a well-defined function of outputs on the input data and the events that arrive from the environment through the event generators.

An FPPN model architecture is defined and tested according to ZD semantics using the TASTE toolset, as it is described in Section 8. Then, as shown in Figure 1, a transformation of the TASTE model takes place, using our tools, into a model in an extension of the BIP language [4] used for modeling tasks [21]. BIP has a well-defined formal semantics as a timed transition system of the produced model given as a network of communicating timed automata (TA) components. After the translation in BIP, the application's structure and model's readability are retained, so that the BIP representation is amenable to manual modification, *e.g.,* as we already mentioned it is possible to define a custom event generator. Moreover, the BIP model is incremental, since it is possible to plug additional components to refine its behavior; we use this feature to optionally plug a scheduler component, when a timing-verified implementation is to be derived. Since any BIP model can be compiled and executed in real time on multi-cores, on top of the BIP RTE (run-time) engine, this model is actually an implementation of the system under design.

As shown in the right-hand side of Figure 1, a design flow extension is provided, for generating a schedulability-verified BIP implementation. The additional design steps auto-generate an incremental extension of the BIP network, called scheduler, and 'plug' it into the BIP model. The purpose is to ensure schedulability of the FPPN on a multi-core platform. As indicated in Figure 1, the FPPN scheduling can take place, when certain, not severe, restrictions (Section 8) are fulfilled to allow an over-approximation of the FPPN workload by a model with a higher workload. This model can be represented by a static task graph with the dependencies between tasks in the over-approximated model. The task graph serves as a model to verify schedulability and generate a static schedule using the algorithm in [36]. An important aspect of multi-core scheduling is the treatment of the various sources of interference in the shared multi-core resources. The scheduling algorithm that we use treats interference by assuming that it can be separated in coarse blocks. Another scheduling framework similar to ours [21] treats the more general case of fine-grained interference (under certain assumptions). However, interference is still an open problem beyond the scope of the present paper.

The BIP implementation represents the RT semantics of the FPPN, where the order of tasks is relaxed within certain limits. Tasks can execute in parallel and the execution of each task is not instantaneous, but it takes a certain delay assumed to be limited by a worst-case execution time. Whereas the overall behavior is not the same with the one prescribed in ZD semantics, a functionally equivalent behavior is ensured by adding special TA components that enforce the proper task order.

The overall approach provides two guarantees, a functional one that is ensured by construction and a timing one that is ensured by verification. The functional guarantee consists of the functional determinism and the functional equivalence between the ZD semantics, i.e. the specification and the RT semantics, i.e. the implementation. The timing guarantee is the verified schedulability, subject to the limitations concerned with the handling of multi-core interference.

## 4 The FPPN Model of Computation

An FPPN model is composed of *Processes*, *Data Channels* and *Event Generators*. The functional code of an application is defined in processes, whereas channels, event generators, and *functional priorities* are necessary

```
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. 2: The functional code of the "SQR" process consisting of subroutines respectively for (i) initialization of internal state and (ii) main process execution.

middleware elements with the latter defining a relation between processes, in order to ensure deterministic execution.

A *Process* represents a software subroutine that has internal variables and input/output channels connected to it through ports. Channels may connect pairs of communicating processes with a data-flow direction (from the writer to the reader) or may interact with the environment (external channels). An example of the functional code of a process ("SQR") is shown in Figure 2.The main functionality of this process is to calculate the square of a valid input value. Its internal state (*index* counter that records the current number of process executions) is firstly initialized by the $SQR\_Init$ subroutine. The SQR process ($SQR\_Execute$ subroutine invocation) reads from the input channel the value of $x$, checks if it *is valid* and computes its square (value of $y$). The write operation on an output channel is consequently performed. A call to the process subroutine ($SQR\_Execute$) 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.

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. A channel is denoted by a $c \in C$ or a pair $(p_1, p_2)$ with a writer and a 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 data-flow or the opposite. Given a $(p_1, p_2) \in \mathcal{FP}$, $p_1$ is said to have a *higher priority* than $p_2$. This relation is necessary to ensure functional determinism: whenever two processes have to access the channel concurrently, the $\mathcal{FP}$ relation defines the order in which the processes access the channel.

The FPPN in Figure 3, represents an imaginary data processing application. A sporadic command from the environment arriving into the input channel invokes process "X" (dashed oval, with timing constraints Deadline=810ms, WCET = 10ms), which is annotated by its minimal inter-arrival time. The "SQR" process (Deadline = 390ms, WCET = 10ms) is periodically invoked to calculate the square of the received value, whereas the "Y" periodic process serves as sink for the squared value (Deadline = 590ms, WCET = 20ms). Periodic processes are annotated by their periods.

Two processes may be connected using one of two possible types of non-blocking channels as those shown in Figure 3. The FIFO (or mailbox) has a semantics of a queue. The blackboard remembers the last written value that can be read multiple times. Additionally, the external input/output channels are shown. Each time that "Y" is called, it receives the squared value from the blackboard channel and sends it to the external output channel.

The data-flow in the channels of this example follows the opposite direction of the functional priority order, which is depicted by the arrow above the channels (its direction illustrates the direction from the higher to lower priority process). A convenient method to define the priority is, by analogy to scheduling priorities, to assign a unique priority index to every process (denoted as 'FP=<index>'), such that the smaller the index the higher the priority it denotes. This method is demonstrated in Figure 3, where the minimal required $\mathcal{FP}$ relation is defined by joining each pair of communicating processes by an arrow going from the higher-priority process to the lower-priority one.

### 4.1 Formal notation and definitions of FPPN entities

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 (resp. vector of do-

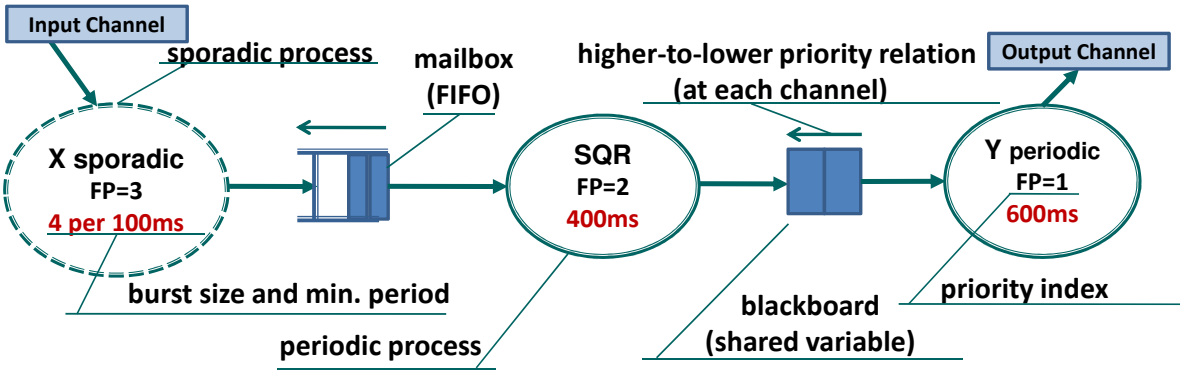


Fig. 3: Example FPPN that is triggered from an external input channel to generate data sporadically, processes it and sends the to an external output channel. A FIFO channel connects the sporadic source with the square process and a shared variable channel connects the square with the sink process. Certain real-time constraints are defined for the processes, while functional priority arrows are depicted above channels.

mains), *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.

Table 2: Use of indices in notations

| Notation | What the indices refer to |
|---|---|
| $\pi^1, \pi^2$ | The valuations of variable $\pi$ |
| $\gamma[1], \gamma[2]$ | The samples of variable $\gamma$ (different instantiations appearing at the first, second, *etc.* time interval) |
| $x?_{[1]}c, x!_{[2]}c$ | The samples of external channel $c$ being read or written |

*Var* includes all *process state* variables $\Pi_p$ and the *channel state* variables $\gamma_c$. The current valuation of a state variable is often simply referred to as *state*. For a variable of channel $c$, an alphabet $\Sigma_c$ 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 $W_c : \mathbf{D}(c) \times \Sigma_c \to \mathbf{D}(c)$ defines the update after writing a symbol $s \in \Sigma_c$ to the channel, whereas $R_c : \mathbf{D}(c) \to \mathbf{D}(c) \times \Sigma_c$ 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 $\gamma_c$ 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 $\epsilon$ 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 $\gamma_c[1], \gamma_c[2], \gamma_c[3], \ldots$, with the same domain. For a sample $\gamma_c[k]$, $k$ is the *sample index*. Though the sequence is infinite, no infinite memory is required locally in the FPPN system, because (as it is later explained) each sample that departs from an external channel to the environment or *vice versa* needs to be kept in a local buffer only for a limited interval of time, until the sample's deadline. If $c$ is an external output, the channel type defines the sample write operation in the form $W'_c : \mathbf{D'}(c) \times \mathbb{N}_+ \times \Sigma_c \to \mathbf{D'}(c)$, 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 $R_c : \mathbf{D'}(c) \times \mathbb{N}_+ \to \mathbf{D'}(c) \times \Sigma_c$. 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 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 $\tau$, $\mathbf{w}(\tau)$.

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

$\alpha = \mathbf{w}(10),\ x?_{[1]}I_1,\ x := x^2,\ x!c_1, \mathbf{w}(100),\ y?c_1,\ O_1!_{[3]}y$

The timestamps in the execution are monotonically increasing. The waiting actions indicate the absolute time of all the data actions that follow until the next waiting action. In the example, at time 10 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 [3] 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 $({\ell_p}^0, L_p, \pi_p, {\pi_p}^0, \mathcal{I}_p, \mathcal{O}_p, A_p, \mathcal{T}_p)$, with $L_p$ a set of locations, ${\ell_p}^0 \in L_p$ an initial location, and $\pi_p$ the set of state variables with initial values ${\pi_p}^0$. $\mathcal{I}_p, \mathcal{O}_p$ are (internal and external) input/output channels. $A_p$ is a set of actions with variable assignments for $\pi_p$, reads from $\mathcal{I}_p$, and writes to $\mathcal{O}_p$. $\mathcal{T}_p$ is a 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 of $X_p$. To ensure that the transition system is indeed deterministic we assume that any two or more transitions branching out from the same location have mutually exclusive guarding conditions.

□

One *execution step* $(\ell_1, \pi^1, \gamma^1) \stackrel{g:a}{\rightarrow} (\ell_2, \pi^2, \gamma^2)$ for the valuations $\pi^1, \pi^2$ of variables in $\pi_p$ and the valuations $\gamma^1, \gamma^2$ 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 $\pi^1$ satisfies guarding condition $g$ (*i.e.,* $g(\pi^1) = True$) and $(\pi^2, \gamma^2) = \mathit{Effect}(a, (\pi^1, \gamma^1))$. Since Def. 1 prescribes a deterministic transition system, for each location $\ell_1$ the guarding conditions enable for each possible valuation $\pi^i$ a single execution step.

**Definition 2 (Process job execution and execution trace)** A job execution is a non-empty sequence of process $p$ execution steps starting and ending in $p$'s initial location $\ell_0$, without intermediate occurrences of $\ell^0$:

$$(\ell^0, \pi^1, \gamma^1) \stackrel{g_1:a_1}{\rightarrow} (\ell_1, \pi_1, \gamma_1) \ldots \stackrel{g_n:a_n}{\rightarrow} (\ell^0, \pi^2, \gamma^2),$$
$$\text{for } n \geq 1, \ell_i \neq \ell^0$$

The projection of a job execution to the actions is called job execution trace, denoted $\alpha$:

$$\alpha = a_1, a_2, ..., a_n \tag{1}$$

A job execution of process $p$ is denoted as:

$$(\pi^1, \gamma^1) \xrightarrow{\alpha}_p (\pi^2, \gamma^2)$$

where $\alpha$ is the job execution trace.

□

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* $EG_p$ that defines the constraints of interaction with the environment. Every $EG_p$ is associated with (possibly empty) subsets $I_p, O_p$ of the external input/output (I/O) channels. Those are the external channels that the process $p$ can access: $I_p \subseteq \mathcal{I}_p$, $O_p \subseteq \mathcal{O}_p$. The I/O sets of different event generators are disjoint, so different processes cannot share external channels.

**Definition 3 (Event)** The term 'event' refers to the time instant $\tau_k$ when two things happen simultaneously: a process $p$ is invoked for the $k$-th time and the samples $\gamma_c[k]$ become available for access in the process's external channels: $c \in I_p \cup O_p$. Formally, we use the term 'event' in two senses: in the narrow sense and in the broad sense.

In the narrow sense an event $e$ is:

$$e = (\tau_k, p)$$

where $\tau_k$ is the invocation time and $p$ is the process being invoked by its event generator.

In the broad sense, an event $e$ is:

$$e = (t, \mathbf{P})$$

where $t$ is invocation time and $\mathbf{P}$ is a (sub-)set of the processes invoked simultaneously by their event generators at time $t$ (whereby different invocations will have, in general, a different sample index $k$). Note $\mathbf{P}$ is a multi-set, since an event generator may invoke a process many times at the same time instant (thus producing a burst of invocations), and each invocation should entail a separate job execution.

□

Every $EG_p$ defines the set of possible sequences of timestamps $\tau_k$ for the event of $k$-th invocation of process $p$ and a relative deadline $d_p \in \mathbb{Q}_+$. The intervals $[\tau_k, \tau_k + d_p]$ determine when the $k$-th job execution can occur. This timing constraint has two important reasons. First, if the subsets $I_p$ or $O_p$ 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, $\tau_k$ defines the order in which the $k$-th job should execute; the earlier it is invoked the earlier it should execute. Concerning the $\tau_k$ sequences, two event generator types are considered, namely *multi-periodic* and *sporadic*. Both are parameterized by a burst size $m_p$ and a period $T_p$. Bursts of $m_p$ periodic events occur at 0, $T_p$, $2T_p$, etc. For sporadic events, at most $m_p$ events can occur in any half-closed interval of length $T_p$. In the sequel, we associate the attributes of an event generator with the corresponding process, *e.g.,* $T_p$ and $d_p$.

**Definition 4 (FPPN)** An *FPPN* is a tuple $\mathcal{PN} = (P, C, \mathcal{FP}, EG_p, I_p, O_p, d_p, \Sigma_c, CT_c)$, 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. This relation should be defined at least for processes accessing the same channel, *i.e.,* $(p_1, p_2) \in C \Rightarrow (p_1, p_2) \in \mathcal{FP} \vee (p_2, p_1) \in \mathcal{FP}$. $EG_p$ maps every process $p$ to a unique event generator, whereas $I_p$ and $O_p$ specify the respective partitions of the global set of external input channels $I$ and output channels $O$, resp. $d_p$ defines the relative deadline for accessing the I/O channels of generator $EG_p$, $\Sigma_c$ defines alphabets for internal and external I/O channels and $CT_c$ specifies the channel types. □

The priority relation $\mathcal{FP}$ 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.

## 5 FPPN Zero-delay Semantics

The functional determinism requirement prescribes that the data sequences and timestamps in the outputs are a well-defined function of the data sequences and time stamps in 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 uni-processor 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. 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 timestamps when the tasks are invoked and secondly from the functional priorities. In this section, we define these constraints in terms of a sequential execution order, i.e. an execution trace defined as the sequence of actions observed when running a sample execution of the system for a given sequence of events (arriving via event generators) and input data (arriving via external input channels).

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 invocation events can only occur at clock ticks, synchronously. Two variant semantics are then defined, namely the *zero-delay (ZD)* and the *real-time (RT)* semantics.

The ZD semantics imposes an order on 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 of events $(t_1, \mathbf{P}^1)$, $(t_2, \mathbf{P}^2)$ $\ldots$, where $t_1 < t_2 < \ldots$ are timestamps and $\mathbf{P}^i$ is the multi-set of processes invoked at time $t_i$. The execution trace has the form:

$$Trace(\mathcal{PN}) = \mathbf{w}(t_1) \circ A_1 \circ \mathbf{w}(t_2) \circ A_2 \ldots \quad (2)$$

where $A_i$ is concatenation of job execution traces of the processes in $\mathbf{P}^i$. The job executions are included in an order, such that if $(p_1, p_2) \in \mathcal{FP}$ then the job(s) of $p_1$ execute earlier than those of $p_2$.

**Definition 5 (Configuration)** An FPPN configuration $\xi$ is a tuple $(\pi, \gamma, \mathbf{P})$ which consists of:

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

□

When the set of pending processes is not empty, the ZD semantics selects the highest priority process, executes its job and removes the process from the list of pending processes. We refer to this operation as dispatching a process. To formally define the dispatching

of a process, as well as other operations of the FPPN semantics, we use inference rules based on the Structured Operational Semantics (SOS) notation, where the proposition above the 'solid line' *i.e.,* nominator is the premise, while the proposition below it *i.e.,* denominator is the conclusion; such a rule is read as follows: if the premise holds then conclusion holds as well.

**Definition 6 (Dispatching a pending process)** Dispatching a pending process is defined as a transition '$\xi \xrightarrow{\alpha}_{\mathcal{PN}} \xi'$' from configuration $\xi$ to configuration $\xi'$, given by the rule:

$$\frac{p \in \mathbf{P} \wedge (\pi_p, \gamma) \xrightarrow{\alpha}_p (\pi', \gamma') \wedge \nexists p' \in \mathbf{P} : (p', p) \in \mathcal{FP}}{(\pi, \gamma, \mathbf{P}) \xrightarrow{\alpha}_{\mathcal{PN}} (\pi \, [\![\pi_p \mapsto \pi']\!], \gamma', \mathbf{P} \setminus \{p\})}$$

where by $\pi[\![\pi_p \mapsto \pi']\!]$ we denote the change in $\pi$ when replacing the state of $\pi_p$ by $\pi'$. Hereby we also recall that $\alpha$ is a job execution trace (see Eq. (1) and '$\xrightarrow{\alpha}_p$' denotes execution of a job of process $p$ (see Def. 2).

□

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

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

Recall the notation $A_i$ introduced earlier, which in fact corresponds to the sequence of data processing actions of a maximal execution run:

$$A_i = \alpha_1 \circ \alpha_2 \circ \ldots$$

$A_i$ always follows immediately after $\mathbf{w}(t_i)$ in the system execution trace.

Given an initial configuration $(\pi^0, \gamma^0)$ and a sequence $(t_1, \mathbf{P}^1)$, $(t_2, \mathbf{P}^2)$ ... 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 timestamps.

$$Run(\mathcal{PN}) = (\pi^0, \gamma^0) \overset{\mathbf{w}(t_1) \circ A_1}{\longmapsto}_{\mathcal{PN}(\mathbf{P}^1)} (\pi^1, \gamma^1) \overset{\mathbf{w}(t_2) \circ A_2}{\longmapsto}_{\mathcal{PN}(\mathbf{P}^2)} \cdots$$

The execution trace of a process network is a projection of the process network run to actions, as it is given by Equation (2). This trace contains the waiting actions for each timestamp $(\mathbf{w}(t_1), \mathbf{w}(t_2) \ldots)$ followed by the data processing actions executed at every timestamp. 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 – job executions – that modify each process/channel states are deterministic themselves and are ordered relatively to each other in a way which is determined by the timestamps and the $\mathcal{FP}$ relation. Therefore we can pose the following claim.

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

*Proof* By *reductio ad absurdum*, let us assume that a non-deterministic modification of a state variable in $(\pi, \gamma)$ takes place. Therefore, in this case, there exists an inter-process channel state variable $\gamma_c$ that is concurrently written by two processes $p_1$ and $p_2$. This implies that these two processes are connected by channel $c$, *i.e.,* $(p_1, p_2) \in C$. Recall that the FPPN model requires that: $(p_1, p_2) \in C \implies (p_1, p_2) \in \mathcal{FP} \vee (p_2, p_1) \in \mathcal{FP}$, i.e. processes $p_1$ and $p_2$ are related by relation $\mathcal{FP}$.

Since these two processes execute concurrently and in ZD semantics we assume the processes take zero time this should imply that they execute at the same timestamp. Now recall that in FPPN ZD semantics the processes should perform a maximal execution run before the time can advance to the next time stamp. This implies that $p_1$ and $p_2$ can execute at the same timestamp only under the condition that they are invoked simultaneously.

Thus we have that $p_1$ and $p_2$ are invoked simultaneously and are related by relation $\mathcal{FP}$. This implies that two simultaneous invocations are executed in the order that respects the relation $\mathcal{FP}$. Thus, there is no non-deterministic choice for the order in which the two processes execute, and hence their execution is not concurrent, which contradicts our initial assumption.

□

In the end of this section, let us provide a more concise definition of what exactly is meant by the '*function of the system*' in the case of ZD execution of the FPPN. For any timestamp $t_m$, let $t_0, t_1, ..., t_m$ be the timestamps that occurred no later than $t_m$. Let $\mathbf{P}^0, \mathbf{P}^1, ..., \mathbf{P}^m$ be the respective processes invoked. Recall that the state of all external channels includes, by definition, the complete history of all elements that were present there at all previous time instants. Then, after the execution of all processes, at time $t_m$ the state $\gamma_o(t_m)$ of the external output channels is a function $\gamma_o(t_m) = FUNC_m(\gamma_i(t_m), \mathbf{P}^0, \mathbf{P}^1, ..., \mathbf{P}^m)$ of: *a)* input channel state $\gamma_i(t_m)$, *b)* the contents of $\mathbf{P}^0, \mathbf{P}^1, ..., \mathbf{P}^m$, *i.e.,* which processes have been invoked and in which order. Basically, the functional determinism property means that the outputs calculated by an FPPN depend only on the events and the input data sequences, but not on the scheduling.

To exploit task parallelism, in the RT semantics, the sequential order of execution and the ZD assumption will be relaxed.

## 6 FPPN Real-Time Semantics

In the RT semantics, job executions last for some physical time and can start concurrently with each other at any time after their invocation. Given precedence constraints are respected, which for certain jobs impose the same relative order of execution as in ZD 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 introduces the entities for communication, synchronization, scheduling and is defined by transformation of an FPPN model to an *executable* formal specification language called BIP (which stands for 'Behavior, Interactions, and Priorities').

The real-time version of BIP [4] is used for modeling networks of connected TA components [42]. We adopt the extension in [21], which introduces the concept of *continuous* (asynchronous) transitions, which, unlike the default (discrete) transitions take a certain physical time. Next to the support for tasks via continuous transitions, BIP also supports the urgency in timing constraints, and those are TA features required for adequate modeling and timing verification of data-flow languages [20]. An important characteristic of the BIP language, for implementing the functional code of tasks, is the possibility to specify data actions in an imperative programming language (C/C++).

In Figure 4, we list in the middle the main elements of a BIP model, whereas on the right two connected BIP components are shown, which form a small BIP model. A BIP component is a transition system with more general semantics than the transition system defining an FPPN process (Def 1). We use BIP components to model not only the processes, but also other FPPN entities, *e.g.,* internal channels and event generators. For each FPPN process and the corresponding event generator we respectively generate a pair of BIP components that are shown in the figure.

If we compare the BIP components to their corresponding FPPN processes, then in BIP we find the same elements – the locations, the initial location, the data actions, the data conditions (the same as the 'guarding condition') and the transitions. The list of ports of a BIP component is analogous to the list of channels connected to a process, except that one can use data actions to define various operations on ports, not only the FPPN-like reads and writes. An important difference of BIP ports from the FPPN read/write actions is that BIP ports are *blocking*, *i.e.,* a BIP interaction via port is only possible when all other ports connected to it via connector are ready to make the interaction synchronously in all components. Another difference is that the transitions are now classified as discrete and continuous. The discrete transitions are similar to ZD semantics, i.e. they are (conceptually) supposed to take zero time.

An essential addition in BIP components is the possibility to use special variables called 'clocks'. Their values start at zero and increase linearly in time. It is also possible to set guarding conditions, called timing conditions, to program some actions when the timing value of a clock satisfies a given requirement. A clock can be reset to zero by a timing action. In Figure 4, we illustrate how to program a periodic event generator using timing conditions and actions.

In the FPPN-to-BIP model transformation process a TA component is created for each process, event generator, and arrow in the relation $\mathcal{FP}$. These components are connected by multi-port connectors. For convenience we indicate the ports that are joined to the same connector by the same name, such as 'Start_<process>', 'Invoke_<process>' *etc.*

Figure 4 illustrates how an FPPN process and its event generator are transformed into BIP components and connected together. The process's functional 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, ports are attached to the generated BIP component, *e.g.,* 'XIF_Read(IN x,IN valid)', via which the respective transitions inside the component synchronize and exchange data with other components. In line with Def. 2, 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 Figure 4 is a simplified variant for periodic tasks, whose deadline is equal to the period. In [40], it is also described how we model internal channels and more details are given on event generator modelling.

To ensure a functional behavior equivalent to ZD 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 cases, the relative execution order of these subsets of jobs is dictated by the ZD 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

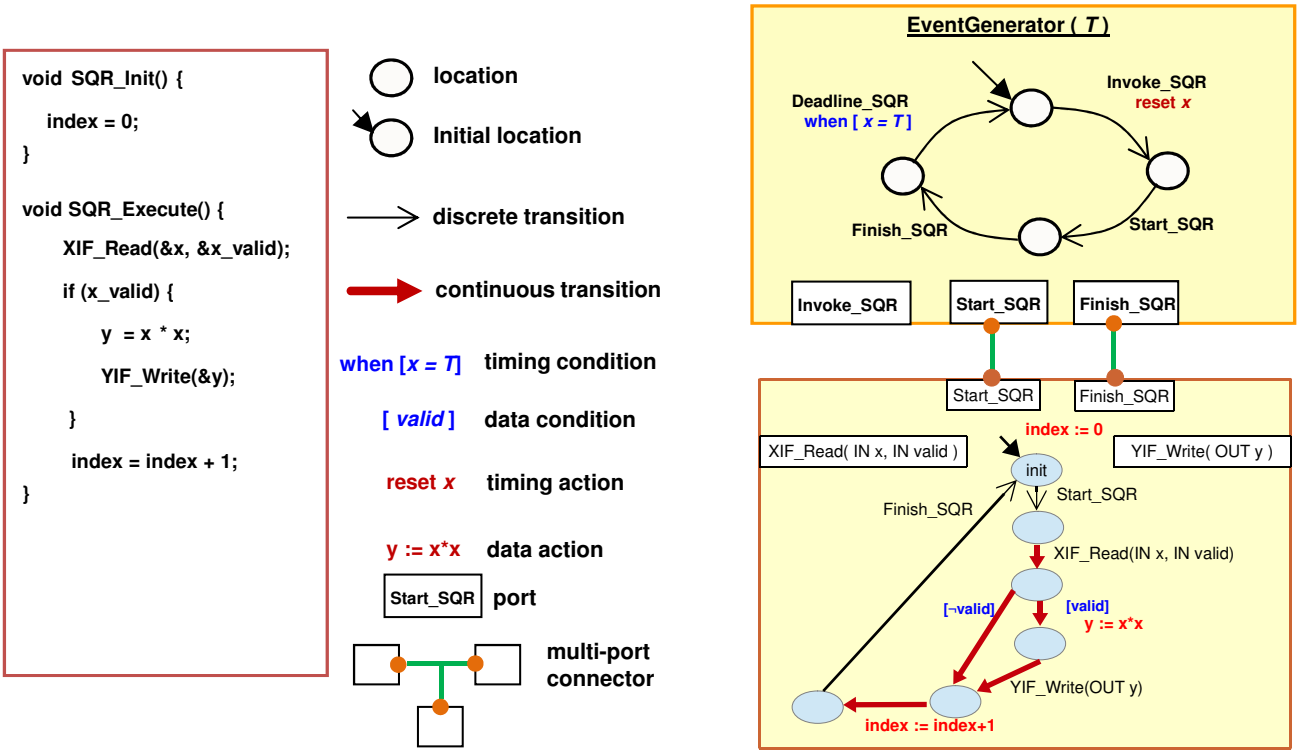


Fig. 4: Transformation of functional code to BIP

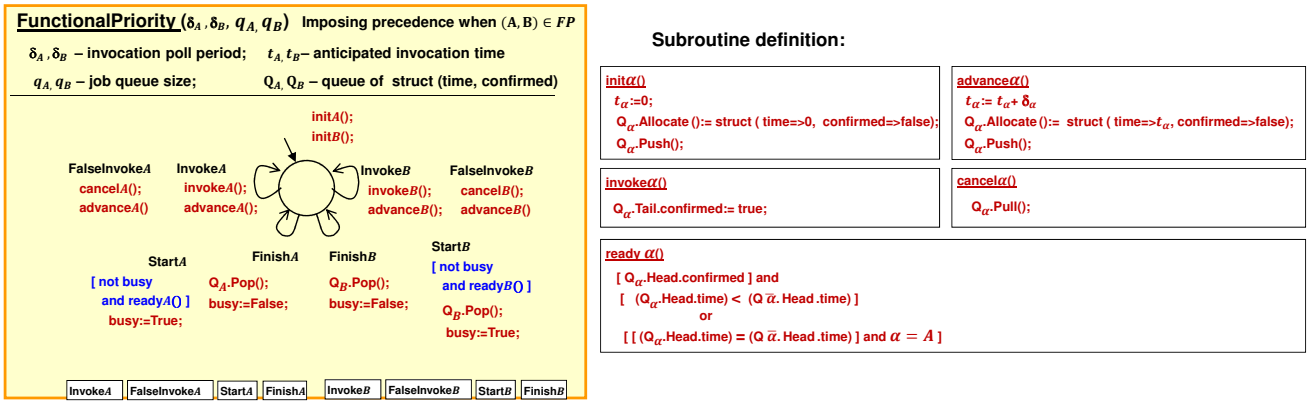


Fig. 5: Functional priority component (FP component) generated for each $(A, B) \in \mathcal{FP}$

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 $\mathcal{FP}$-related processes and plugged incrementally into the network of BIP components.

Figure 5 shows such a component generated for a given pair of processes "A" and "B", assuming $(A, B) \in \mathcal{FP}$. We saw in Figure 4 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 FP component is connected via ports 'Invoke $\alpha$' or 'FalseInvoke $\alpha$' to the event generator (EG) component of the respective processes. At regular intervals $\delta_\alpha$, specific for the process $\alpha$, the FP component performs either 'Invoke' or 'FalseInvoke' interaction, with the decision which of them is selected being taken by the EG component. For periodic processes $\delta_\alpha = T_\alpha$ (*i.e.,* the period) and the decision is always to perform an 'Invoke' interaction (check the periodic EG component in Figure 4). For sporadic processes, typically $\delta_\alpha \ll T_\alpha$ and most of the time only 'FalseInvoke' interactions are performed and only occasionally there are 'Invoke' interactions. We use the name 'sporadic protocol' for a user-defined subroutine that runs inside the EG with period $\delta_\alpha$, polls the environment (I/O peripherals) and decides whether the sporadic process needs to be invoked to handle an external event.

The FP component has two variables of type 'queue' (a FIFO buffer of data entries of certain type). Since the entries of these two queues contain the information about jobs, we refer to them as 'job queues'. A queue is implemented by a circular buffer with the following operations:

- `Allocate`() picks an available (statically allocated) cell and creates a reference to it
- `Push`() pushes the last allocated cell into the *tail*
- `Pull`() undoes the push
- `Pop`() retrieves the data from the *head* of the queue

The two job queues are denoted by $Q_\alpha$ where $\alpha \in \{A, B\}$ indicates to which of the two processes it refers. With $\overline{\alpha}$ we denote 'other than $\alpha$', *i.e.,* if $\alpha = A$ then $\overline{\alpha} = B$ and if $\alpha = B$ then $\overline{\alpha} = A$. The jobs are arranged in a job queue in an order consistent with their invocation times, starting from the earliest-invoked job in the head and ending with the latest-invoked job in the tail. An entry in the queue contains the invocation time and a Boolean flag indicating whether the invocation at a given time has been already 'confirmed' by an 'Invoke' interaction. An invariant of the FP component behavior is that the job queue tail always contains the next anticipated job, which is conservatively marked as 'not yet confirmed'. The older jobs in the queue, closer to the head, are always 'confirmed', otherwise they would not have been kept in the queue. The non-confirmed job at the tail prevents the situation where, after all previous jobs have been executed, the $\overline{\alpha}$ process would execute some job, whereas it has not yet been 'confirmed' whether process $\alpha$ has to execute a job that was invoked at the same time or earlier. The 'Invoke' interaction 'confirms' the job in the tail (via 'invoke_$\alpha$' subroutine) and pushes a new anticipated non-confirmed job (via 'advance_$\alpha$' subroutine). The 'FalseInvoke' interaction removes the last anticipated (but not confirmed) job from the tail (via 'cancel_$\alpha$') and pushes a new one at the next anticipated time. The 'Start' interaction can only take place for a confirmed job in the head of the queue under the condition that the job of the other process has a later invocation time or lower priority. This is checked via the 'ready_$\alpha$' subroutine. When the job has finished executing it is removed from the queue.

## 7 Functional Equivalence of ZD and RT Semantics

We study here the differences and the correctness properties of the ZD and RT semantics, which eventually entail an equivalent functional behavior. Hereby, for the ZD semantics we are based on its formal definition and for the RT semantics we mainly use the properties of the BIP FP component for Functional Priority, as described in previous section and in Figure 5. Recall that for each pair of functional-priority related processes in the FPPN, our FPPN-to-BIP model transformation creates an instance of the FP component that is connected to the BIP components for the two processes.

We will show that both semantics satisfy – by construction – three key correctness properties of the FPPN MoC: (1) mutual exclusion of functional priority related processes, (2) conformance to the invocation times, and (3) conformance to the functional priorities. For the ZD semantics, the properties are implied by the definitions, and for the RT semantics by the properties of the FPPN-to-BIP model transformation. Finally, we will show that the aforementioned properties entail a functional equivalence between the two semantics.

In both semantics, processes are specified in terms of transition systems, as is seen in Def. 1 for the ZD semantics and from the bottom-right BIP component of Figure 4, for the RT semantics. However, the TA transitions may take non-zero time, as opposed to the process definitions for the ZD semantics. Moreover, in BIP, apart from the usual data-computation actions, the automaton may perform actions classified as *interactions via ports*, to synchronize and exchange data with other components. The BIP component for a given process is obtained from the FPPN process automaton through the following transformation steps: (i) the 'Start' and 'Finish' transitions are added for interactions via the respective ports, (ii) the read actions '$x?c$' and write actions '$y!c$' are interpreted as interactions through ports 'Read($c$,$x$)' and 'Write($c$,$y$)', *i.e.,* they do not perform direct accesses to the state '$\gamma_c$' of channel $c$, but instead 'instruct' the data channel components to update their state by the respective interactions. Consequently, in

BIP, the process components interact with the channel component 'Channel($c$)' connected to their 'read' and 'write' ports.

If we abstract from the timing of actions and check their order, the execution traces of an FPPN process or BIP process are virtually the same. Let a job execution of the FPPN be:

$$(\ell^0, \pi^1, \gamma^1) \overset{g_1:a_1}{\rightarrow} (\ell_1, \pi_1, \gamma_1) \ldots \overset{g_N:a_N}{\rightarrow} (\ell^0, \pi^2, \gamma^2),$$

with trace: $\alpha = a_1 a_2 \ldots a_N$. In this case, the trace of executed BIP transitions will be

$$\alpha = \text{Start } a_1 a_2 \ldots a_N \text{ Finish}$$

The three correctness properties are formulated using *logical formulas* that characterize the system's *state trace* that is given as a sequence of global states that the system visits in-between actions. For both semantics the global state includes the current values of state variables and the current locations '$\ell$' of all process automata. Additionally, for the ZD semantics, it also includes the current configuration and the current time (*i.e.,* the timestamp of the last $\mathbf{w}(t)$ action). For the RT semantics, on top of the process automata related state, the global also includes the variable state and locations of the other BIP components, *i.e.,* the data-channel components, the EG components, and the FP components. For our logical formulas only the following information in the system state is relevant:

- whether or not a given process automaton is in its initial location $\ell^0$
- what is the latest invocation time of a given process; for the ZD semantics it simply equals to the current time, and for the RT semantics it can be derived from the state of automata, in particular, it is contained in the head of the respective job queue in a related FP component.

The logical formulas for the correctness properties are, in fact, theorems for both the ZD and RT semantics and our arguments for their validity indicate the lines of reasoning for their proofs. At the top-level, the formulas quantify over processes $p$ and time variables $t$. The predicates are the LTL formulas, relations between processes, and (in-)equalities over time variables. The LTL formulas express statements about the system state trace and they use the common operators *i.e.,* the always (true now and forever) '$\Box$' operator and the eventually or finally (true now or sometime in the future) '$\Diamond$' operator. To simplify the notation we omit all quantizers from the formulas (if necessary they can be easily guessed).

Regarding the the mutual exclusion property, first, we note that in ZD semantics there are no two job executions that may be ever interleaved with each other in the FPPN system action trace, *i.e.,* there are no FPPN action fragments like $a_1 a'_1 a'_2 a'_3 a_2$, where $a_i$ and $a'_i$ are steps of two different job executions $\alpha$ and $\alpha'$. Therefore, the system execution trace is *defined* as concatenation of complete job executions: the FPPN behavior is seen as selecting a process, letting it run a complete sequence of steps of a job execution until the return to its initial state $\ell^0$, and then selecting another process and repeating the same cycle, and so on. On the other hand, the BIP operational semantics [7] *a priori* lets all components execute their actions concurrently, except those components that are blocked waiting for interactions. Consequently, the RT semantics potentially allows some interleaving.

The non-interleaving behavior of the ZD semantics prevents data race conditions between functional-priority related processes, where races could occur in the access order to the shared data channels. For any two processes that do not share a data channel, their interleaving does not impact the functional behavior, and thus can be permitted in RT semantics.

We conclude that the RT semantics has to fulfill the *mutual exclusion* correctness property, which delimits the necessary non-interleaving.

**Mutual Exclusion of functional priority related processes** $\quad p \bowtie p' \;\Rightarrow\; \Box(\neg \ell_p^0 \;\Rightarrow\; \ell_{p'}^0)$
where by $p \bowtie p'$ we denote two processes that are functional priority related ($(p, p') \in \mathcal{FP} \vee (p', p) \in \mathcal{FP}$) and $\ell_p^0$ states that at the beginning of trace the automaton of $p$ is at its initial location $\ell^0$, and hence no job execution of that process is currently active.

The operator '$\Box$' (referred to as 'always') means an assertion that the logical statement to which the operator is applied is true for the system trace, if we start capturing it from no matter which moment of time. The property specifies that if two processes are functional priority related then whenever a job execution of one process is currently running no job execution of the other process is also running at the same time. This means that the processes concerned do not interleave their job executions.

According to the RT semantics, the BIP automata for the two functional-priority related processes are connected to an FP automaton. As we see in Figure 5, whenever one of the processes starts a job execution – at the 'Start' interaction – the $\mathcal{FP}$ component sets its 'Busy' flag to 'true', which prevents that another 'Start' interaction is initiated before the execution of the 'Finish' interaction. The FP component is therefore guaranteeing mutual exclusion between the given pair of processes. Since the ZD semantics satisfies mutual exclu-

sion by construction, we conclude that both semantics satisfy this property.

The next property requires the system to let the job executions of functional priority related processes start in an order that respects the invocation time:

**Conformance to Invocation Order** $p \bowtie p' \;\Rightarrow$
$\Box\left(\neg\ell_p^0 \wedge it_p = t \wedge \Diamond(\neg\ell_{p'}^0 \wedge it_{p'} = t') \;\Rightarrow\; (t \leq t')\right)$
where $it_p$ is a function of the system state that returns the invocation time of the latest job execution for $p$ that has started so far.

Operator '$\Diamond$' (referred to as 'eventually') states that the logical formula to which the operator is applied will eventually hold in the future. Thus, the property states that the invocation time of a job that is currently running, does not exceed the invocation time of any job that starts later.

This property is satisfied by the ZD semantics, because at every moment $t$ the maximal execution of all jobs invoked at $t$ is performed, whereupon any new job can be invoked only at a moment $t'$ later than $t$. For the RT semantics, let us consider a job that is 'ready' to 'start' according to the FP component, shown in Figure 5. Since the queue structures contain the job invocation times, the $ready\_\alpha()$ condition states that if two jobs have different execution times only the job that is invoked earlier can be ready to execute first.

The following property determines the order of jobs that are invoked simultaneously:

**Conformance to Functional Priority** $p \bowtie p' \;\Rightarrow$
$\Box\left(\neg\ell_p^0 \wedge it_p = t \wedge \Diamond(\neg\ell_{p'}^0 \wedge it_{p'} = t) \;\Rightarrow\; (p, p') \in \mathcal{FP}\right)$

The ZD semantics satisfies this property based on the definition of '$\xrightarrow{\alpha}_{\mathcal{PN}}$' transition relation between two configurations, which excludes the possibility that a process may execute a job while a higher-priority process is pending. By inspecting the '$ready\_\alpha()$' condition, it is obvious that the RT semantics also satisfies the property.

The mutual exclusion property excludes the data races at internal channels at the level of individual actions, and the two other properties ensure that there are no races at the level of job executions, since the job order is a function of the jobs' invocation timestamps. The functional determinism theorem and its proof show that this order of job executions determines a unique function of the system outputs on the system inputs. Thus, the two semantics exhibit unique and hence equivalent functional behavior.

## 8 Design Framework

### 8.1 Design Flow Toolchain

The model-based design philosophy for embedded systems [29] that we follow is grounded on the principle of evolutionary design using models, which support the gradual refinement of system's design (making the models more detailed and accurate) including the configuration of real-time attributes that ensure predictable timing and schedulability. Such a process allows considering various design scenarios and promotes the late binding to design decisions. Our approach to refinement is based on component-based models, which allow the design to evolve incrementally by plugging new components and transforming existing ones.

In the design flow of Figure 1 in Section 3, we take as a starting point a set of tasks defined by their functional code and real-time attributes (*e.g.,* periods, deadlines, WCET). We assume that these tasks are encapsulated into a software architecture with '*functional blocks*' – or, simply, '*functions*'. Functions correspond to FPPN processes. Before being integrated into a single architectural model they can be compiled and tested separately by functional simulation or by running them on the embedded platform.

Capturing the design is facilitated using tools with a GUI interface, whereas the design flow is automated using scripts and analysis tools. The entire design framework, called TASTE2BIP, is available for download at [3]. The high-level architecture description framework that supports our design flow is the TASTE toolset [29, 35], whose front-end tools are based on the AADL (Architecture Analysis & Design Language) syntax [16]. TASTE is an extensible open-source software engineering framework for real-time distributed system design. It is amenable to customization by new code generation tools and languages. For our design flow, we are based on the translation of FPPN networks in TASTE to BIP, as an intermediate step for the real-time scheduling on *multiple processor cores* with functionally deterministic communication and schedulability constraints.

An *architecture model* in TASTE consists of functional blocks, which interact with each other via pairs of interfaces (IF), the so-called 'required IF' and 'provided IF', where the former performs a procedure call to the latter. The provided interfaces can be explicitly used for invoking tasks, *i.e.,* they may get attributes like 'periodic'/'sporadic', 'deadline' and 'period'. Each FPPN process is represented by a unique functional block that 'provides' a unique sporadic or periodic interface, whose implementation in C/C++ language defines a job execution. The functional block also 'requires' reading and

writing interfaces for each data channel that the process reads (resp. writes) from (resp. to). Also, each process is assigned a unique integer, a defined TASTE attribute that corresponds to the functional priority $\mathcal{FP}$ index. Higher $\mathcal{FP}$ indexes refer to lower priorities, as defined in Section 4 (cf. Figure 2).

Moreover, criticality levels can also be defined for the processes, according to their importance for safety of people and equipment. For example, in avionics, criticality A is given to the computers that control the flaps of the wings, criticality B to autopilot, criticality C to communication with the ground, criticality E for entertainment on board. Safety standards define respective tolerated error rate per hour of flight, $10^{-9}$ for autopilot and usually $10^{-3}$ for the entertainment of board (each level is associated with an integer *i.e.,* A = 5, B = 4, C = 3, D = 2, E = 1). The criticality levels are taken into account by our offline scheduling tool [36]. The proposed flow currently supports two levels of criticality *i.e.,* 'D' and 'E', where 'D' corresponds to process with higher criticality level than 'E'.

The first refinement step in Figure 1 comprises the connection of processes with each other by 'plugging' the data channels for explicit communication between the processes. A data channel is also represented by a single TASTE functional block, which 'provides' one read and one write interface, being plugged into the 'required' interfaces of the TASTE blocks for the processes that read and write to them. By editing the TASTE attributes of the functional blocks, we can specify the type of data channel (*i.e.,* mailbox or blackboard) and its capacity (for a mailbox), in terms of the number of data items that are buffered in the channel. A temporary required attribute of a data channel is the (upper bound of) byte size of one data item, which in fact is a platform-dependent low-level detail that can be deprecated in future versions of our framework, as TASTE supports more powerful data type management[2]. The attributes of TASTE functional blocks have been amended as appropriate, in order to reflect the priority index and criticality of the FPPN processes, as well as the channel parameters. The resulting model can be compiled and simulated within the TASTE native framework, as well as in our design framework too.

[2] To this end, TASTE supports a language called ASN.1 in which we can define the data types for all 'provided'/'required' interfaces in a platform-independent manner. Moreover, an ASN.1 compiler is also provided that translates the data type specification into platform-dependent code and ensures appropriate cross-platform data conversion. In our framework, such capabilities are future work prospects, since we currently focus on a single multi-core platform and no support is provided for distributed systems with communication between different platforms.

The next refinement step is the scheduling for the multi-core platform. This step includes the transformation of the TASTE architectural model into the equivalent BIP model for the FPPN network (Figure 1). Through this, the FPPN network can be coordinated by the multi-core BIP RTE, which guarantees that the formally defined RT semantics for the network of BIP components is respected, while they are executing on multiple cores. Thus, we provide a formal, incremental component-based model for scheduling, while respecting by construction the RT semantics of FPPNs.

The scheduling decisions are computed offline, using a static mapping of processes to cores and a static ordering policy, for which the scheduler has to compute the optimal order to be used online. The scheduling decisions are subsequently reflected by plugging an additional BIP component, called *online-scheduler* (*cf.* Figure 6), which, on top of the FP components, further restricts the order and timing of execution of the process components on each processor core. The offline computation of the static schedule is based on a task graph, which is automatically generated from the architectural model (Figure 1) by taking into account the periods, the functional priorities and the WCET of processes. The WCET should be obtained by analysis tools for the code derived from the BIP model with the BIP compiler and linked with the BIP RTE. The task graph represents a maximal set of jobs invoked in a hyperperiod (least common multiple of all periods) and their precedence constraints; it reflects the invocation times and the deadlines of jobs relatively to the hyperperiod start time.

For deriving a task graph that correctly represents the sporadic processes, we adopt certain restrictions for these processes in the FPPN network [38]. In particular, every sporadic process $p$ should be connected by data channels to a single periodic process, referred to as 'user' process $u(p)$. A sporadic process should not be related by $\mathcal{FP}$ to any other process than its unique periodic 'user', which makes it possible to reflect its precedence constraints in a task graph. This restriction also makes sense from a practical point of view, as the sporadic tasks often act as utility to dynamically re-configure some of the parameters of certain periodic tasks, which are thus 'users' of sporadic tasks according to this terminology.

The task graph is given as input to the static scheduler, while the new schedule obtained from the tool is manually translated into constraints for the online-scheduler (this step in Figure 1 has not yet been automated). First, by inspecting the time charts of the generated offline schedule, the user has to define a textual file for the static mapping of process components to the cores. Secondly, for those processes that are mapped to the same core and are not functional priority related,

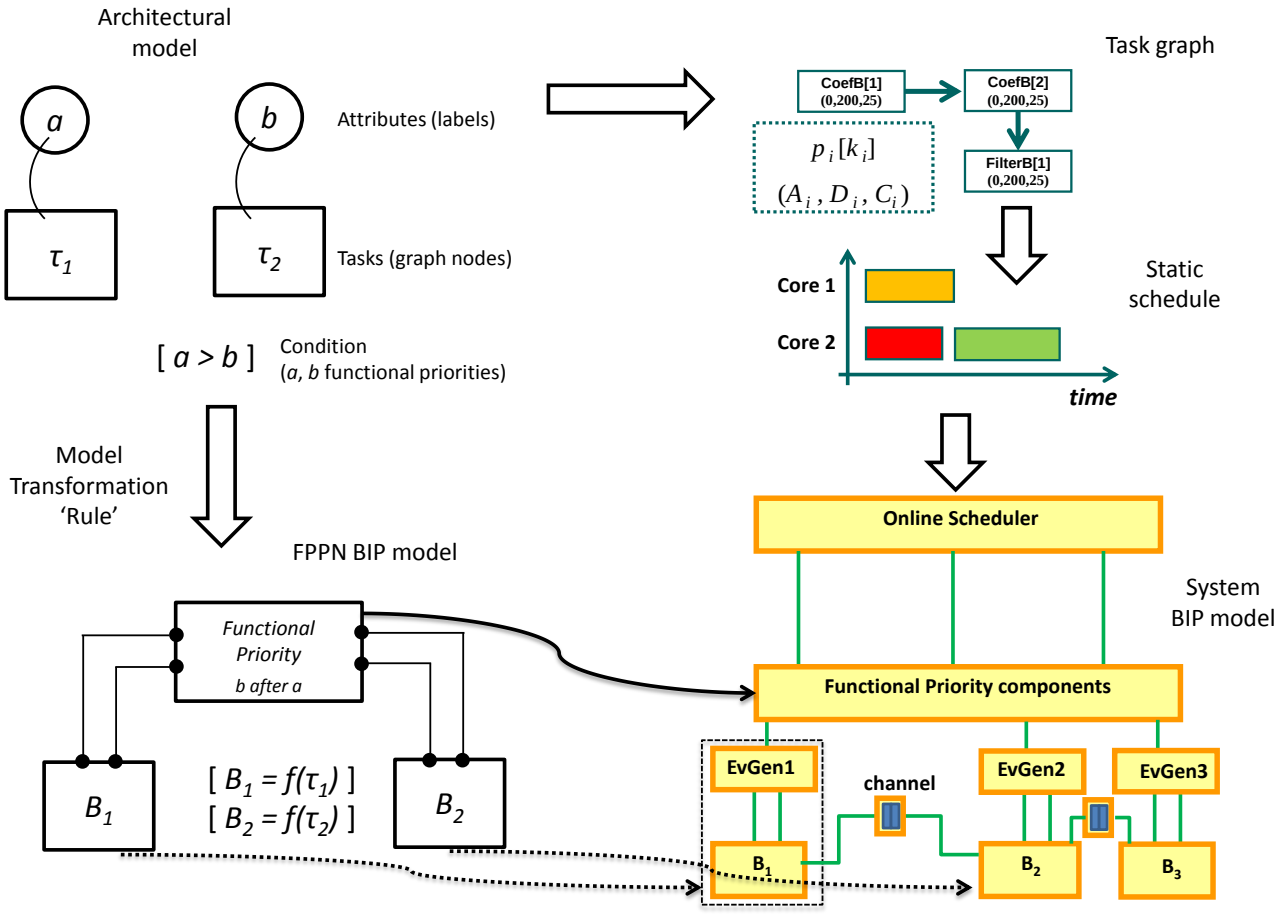


Fig. 6: Model and graph transformations for the FPPN semantics

new FP components have to be added, which ensure the same order as the static order computed by the tool. These additional FP components altogether, in fact, constitute the 'online scheduler'. Note that one can either add new edges to the $\mathcal{FP}$ relation in the high-level FPPN representation before FPPN-to-BIP transformation or plug new instances of FP component into the BIP model of FPPN.

The joint application and scheduler model is called System Model (Figure 1). This BIP model is eventually compiled and linked with the BIP-RTE for the final implementation on the embedded platform.

### 8.2 FPPN2BIP Model Transformation

The model transformation tool maps the TASTE FPPN model entities (XML file and C/C++ code) to BIP as shown in Table 3 and eventually generates an equivalent BIP model according to the RT semantics. The model transformation, which is depicted in Figure 6, consists of: (a) code-to-code transformations of TASTE-embedded code into BIP-embedded C/C++ code, and (b) graph-to-graph transformations, where a network of TASTE components is transformed into a network of BIP components. It can be described by the following set of transformation rules that synthesize the network of BIP components according to the configuration of the FPPN model:

(i) *Event Generator and BIP Process Constitution:* Every task $\tau_a$ is transformed into a BIP subnetwork $B_a$ comprising a BIP Task component and an Event Generator component $EvGen_a$ bound to each other by two connectors, $Start_a$ and $Finish_a$, as shown in Figure 4.

(ii) *Connection between Task and Functional Priority Components:* Each BIP subnetwork $B_a$ is connected to the FP Components that correspond to the $\mathcal{FP}$ arrows, which involve $\tau_a$. The aforementioned $Start_a$ and $Finish_a$ connectors are extended to include the

Table 3: TASTE2BIP model transformation

| **FPPN entity** | **TASTE representation** | **Transformation** | **BIP component(s)** |
|---|---|---|---|
| Periodic process | *Function with attributes*:<br>·Class=periodic_process<br>·Fpriority=integer<br>*Provided interface:*<br>·Kind=cyclic<br>·period, deadline, WCET | Function to Task component<br>+ Event Generator | Task and<br>Event Generator with<br>periodic features<br>and invocations |
| Sporadic process | Two functions†<br>*Function #1 attributes*:<br>·Class=sporadic_process<br>·Fpriority=integer<br>*Provided interface for #1:*<br>·Kind=sporadic<br>·min-interarrival,deadline,<br>WCET, queue size<br>*Function #2 attributes*:<br>·Class=sporadic_protocol<br>*Provided interface #2:*<br>·Kind=cyclic<br>·protocol period $\delta$ | Function #1 to Task component<br>and Function #2 to Event Generator<br>instrumented by given<br>sporadic protocol | Task and<br>Sporadic Event Generator |
| Blackboard<br>channel | *Function with attributes*:<br>·Class=blackboard<br>·DataChannelSize (size of<br>ASN.1 data type)<br>**write port**⋆parameters:<br>IN data, valid,<br>OUT write_fail<br>**read port**⋆parameters:<br>OUT data, valid | Function to<br>component implementing<br>Blackboard (shared variable) | Blackboard channel<br>with the given<br>data item size |
| Mailbox (FIFO)<br>channel | *Function with attributes*:<br>·Class=mailbox<br>·DataChannelSize (size of<br>ASN.1 data type)<br>·DataChannelLength=int<br>**write port**⋆parameters:<br>IN data, valid,<br>OUT write_fail<br>**read port**⋆parameters:<br>OUT data, valid | Function to<br>component implementing<br>Mailbox (FIFO) | Mailbox (FIFO)<br>channel with the given<br>data item size<br>and length (number of<br>items) |

† Required interface of function #2 connected to provided interface of function #1.
⋆ Ports connected with provided interfaces, assigned attributes of connected processes.

respective ports of the relevant FP components. The Event Generator component $EvGen_a$ is also bound by connectors $Invoke_a$ and $FalseInvoke_a$ to the respective ports of the FP components. In the end, if there are two functionally priority related tasks $\tau_1$ and $\tau_2$, with priority indexes $a$ and $b$, respectively, where $a > b$, then their Task components will be plugged to the FP component corresponding to the arrow $(\tau_2, \tau_1)$ in relation $\mathcal{FP}$ (Figure 5).

(iii) *Connection of Task with Channels:* If two tasks $\tau_1$ and $\tau_2$ write and read data to/from an internal channel $c$, then their Task component is connected via $Write$ and $Read$ ports to the Channel component $Channel(c)$. A Task component can have multiple $Write$ and $Read$ ports depending on the number of the channels it connects to.

### 8.3 Task Graph Generation

Scheduling with precedence constraints is usually based on task graphs, which exist in different flavors. The Task Graph, as defined below, represents the process job execution of an FPPN network during one hyperperiod. Since the system's hyperperiod is continuously repeated, it suffices to schedule one hyperperiod and then repeat the resulting schedule periodically.

**Definition 7 (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, $k_i \in \{1, 2, 3, \ldots\}$ 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 $C_i \in \mathbb{Q}_+$ is the WCET. The $k$-th job of process $p$ is denoted by $p[k]$. The edges $\mathcal{E}$ represent the precedence constraints.

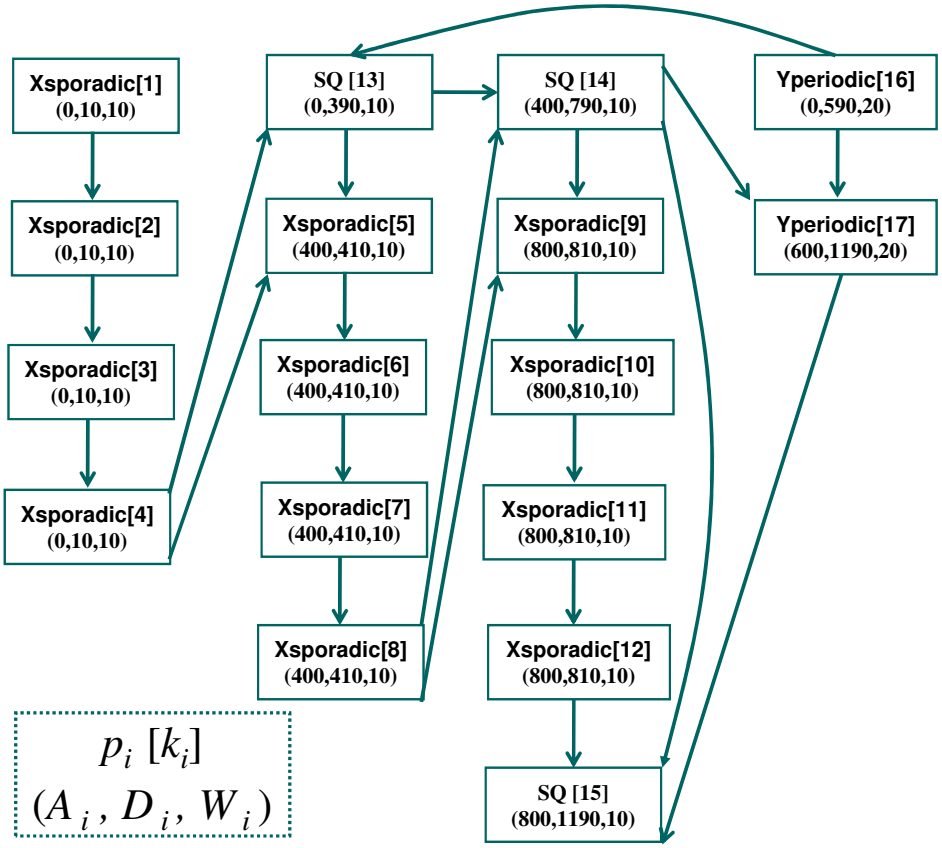


Fig. 7: The Task Graph generated for the FPPN network example shown in Fig. 3

The invocation times, deadlines and job invocation counts are specified to be relative to the beginning of the hyperperiod.

The sporadic processes introduce uncertainty for the invocation times of tasks, which in principle complicates the scheduling. As mentioned before, we consider that every sporadic process $p$ is functionally related to one and only one 'user' process $u(p)$, which allows to compute the hyperperiod of the system. To cope with sporadic jobs invoked inside the user period, we represent a sporadic process by an '$m$-periodic server' task, whose jobs that are invoked at the boundaries of the user period intervals represent the jobs that have been invoked in the preceding user-period interval. For convenience, we require that the user period should not be greater than the inter-invocation interval of the sporadic process: $T_{u(p)} \leq T_p$. If $m$ is the burst size (the maximal number of invocation events per $T$) of the sporadic process, by the above constraint, the burst size of the periodic server can be conservatively selected to be equal to $m$ as well.

Algorithm 1 describes the procedure for generating the task graph from a given FPPN process network $\mathcal{PN}$. First, the algorithm obtains a new process network $\mathcal{PN}'$, where all sporadic tasks $p$ are replaced by their periodic server $p'$. We use 'prime' for the parameters of network $\mathcal{PN}'$, e.g., $d'$, $T'$. Note that every burst of $m$ periodic server jobs, by definition, represents the sporadic jobs invoked during the period *before* the respective user job is invoked at the end of the interval. Being invoked before the user job, the sporadic jobs should have, according to FPPN semantics, precedence over the user job, and it is for this reason that we set the priority of the server to be higher than that of the user: $(p', u(p)) \in \mathcal{FP}'$. Additionally, the deadlines of the server jobs are conservatively corrected according to the formula $d'_{p'} = d_p - T_{u(p)}$. This correction is required due to the fact that in our framework the periodic server lets a newly invoked spo-

radic job wait, for at most $T_{u(p)}$, until the end of the period, and only then it is considered as an invoked periodic-server job. Hence, the worst-case waiting time has to be subtracted from the relative deadline.

**Algorithm 1** Task Graph Generation

**Input:** FPPN Model $\mathcal{PN}$ (in TASTE XML representation).
**Output:** Textual representation of a Task Graph

1: Generate a new FPPN model $\mathcal{PN}'$, where each sporadic process $p$ is replaced by $m$-periodic 'server process' $p'$ with burst size $m'_{p'} = m_p$, period: $T'_{p'} = T_{u(p)}$, priority relation: $(p', u(p)) \in \mathcal{FP}'$ and deadline: $d'_{p'} = d_p - T_{u(p)}$.
2: Calculate the least common multiple $LCM$ (hyperperiod $\mathcal{H}$) of the process periods in $\mathcal{PN}'$.
3: By simulation of zero-delay execution for process network $\mathcal{PN}'$ generate the trace $J$ of the jobs executed in one hyperperiod. The generated trace should be of the form $J = (p_i[k_i])$ (process identifier and its invocation count). Let $<_J$ the total order defined by the trace.
4: Construct $\mathcal{TG}(\mathcal{J}, \mathcal{E})$, where nodes $\mathcal{J}$ are the elements of the generated sequence $J$ and the edges $\mathcal{E}$ between jobs denoted as $J_a = p_a[k_a]$ and $J_b = p_b[k_b]$ are defined as follows:
   (i) $(J_a, J_b) \in \mathcal{E} \iff J_a <_J J_b \wedge (p_a \bowtie p_b \vee p_a = p_b)$ with $p_a \bowtie p_b \iff (p_a, p_b) \in \mathcal{FP}' \vee (p_b, p_a) \in \mathcal{FP}'$, which means that the processes are related by $\mathcal{FP}'$ and job $J_a$ is executed earlier than job $J_b$ in the sequence $J$.
   (ii) Job parameters for job $J_i = p[k]$:
   – $A_i = T'_p \cdot \lfloor (k-1)/m'_p \rfloor$
   – $D_i = A_i + d'_p$
5: Truncate all the deadlines $D_i$ to the hyperperiod: $D_i := min(\mathcal{H}, D_i)$.
6: Perform transitive reduction of relation $\mathcal{E}$ to remove redundant edges.

Subsequently, at Step 3 the algorithm performs simulation of one hyperperiod of the multi-periodic network $\mathcal{PN}'$ to generate a sequence of job executions in the order respecting the FPPN ZD semantics. This can be done in a straightforward way by applying the definition of the ZD semantics.

At Step 4, we create a graph whose nodes are jobs in the generated sequence and whose edges are generated as follows. A pair of jobs is joined by an edge if they belong to the same process or to functionally priority related processes. The edge direction respects the order of jobs in the job sequence. The invocation times and the absolute deadlines are derived from the invocation count $k$, process's period $T_p$ and relative deadline $d_p$.

At Step 5, since we assume constrained-deadline scheduling, all jobs should finish by the end of the hyperperiod. Thus the deadlines of all jobs are truncated to the hyperperiod duration.

At Step 6, the graph is simplified to remove redundant edges by transitive reduction, *i.e.,* obtaining the minimal relation whose transitive closure is the same as the transitive closure of the original relation. This can be applied because the 'precedence' relation between jobs, which is represented by the task graph edges, is a transitive relation.

Figure 7 depicts the derived Task Graph for the FPPN network of Figure 3. In this figure we see three occurrences of a chain of four subsequent sporadic jobs 'XSporadic$[4k+1, 2, 3, 4]$' that precede a job 'SQ$[k]$'. This is because the process 'XSporadic' in Figure 3 has burst size four and its user is process 'SQ'.

### 8.4 Task Scheduling

To present the main principles of FPPN scheduling let us consider the illustrative example in Figure 8 with three tasks. The 'split' task appends two small data items (a few bytes) in two output channels and sleeps for 1 ms to imitate some task execution time. Tasks 'A' and 'B' read the data and Task 'A' sleeps for 12 ms whereas Task 'B' sleeps for 6 ms. All tasks have the same periodic scheduling window, with period and deadline being 25 ms. In the derived task graph, every task is represented by a job. The jobs are numbered as $J_i = J_1, J_2, J_3$ and annotated by their WCETs. The arrival times $A_i$ and deadlines $D_i$ for all jobs are the same.

The static scheduler accepts a parameter $\epsilon$ for the worst-case cost of a single transition in the BIP TA components. Parameter $\epsilon$ is platform-dependent and characterizes the interference between the task components, when they access the BIP RTE to execute discrete automata transitions. It defines the worst-case execution time that it takes for the BIP RTE engine to coordinate the execution of one discrete interaction of BIP components. This parameter is calculated by measurements on the the target platform and back-annotated to the static scheduler. If there are no available measurements for parameter $\epsilon$, the user can start with an estimation of $\epsilon$ parameter and adjust it later in the design cycle. Before applying the static scheduling tool to the task graph, the graph is automatically extended by inserting for every application job four other jobs, called virtual jobs, having WCET $\epsilon$ and representing the four discrete BIP interactions involved in a job execution: 'Invoke', 'Start', 'Finish', and 'Deadline' [36]. Based on the extended task graph, the total maximal system load (*i.e.,* resource utilisation factor) is automatically calculated, which indicates the minimal number of CPU cores needed.

The scheduling tool applies list scheduling algorithm based on an heuristically computed fixed priority relation (consistent with the $\mathcal{FP}$ relation). For the given number of CPU cores, the tool distributes the jobs between the cores and determines their start times based

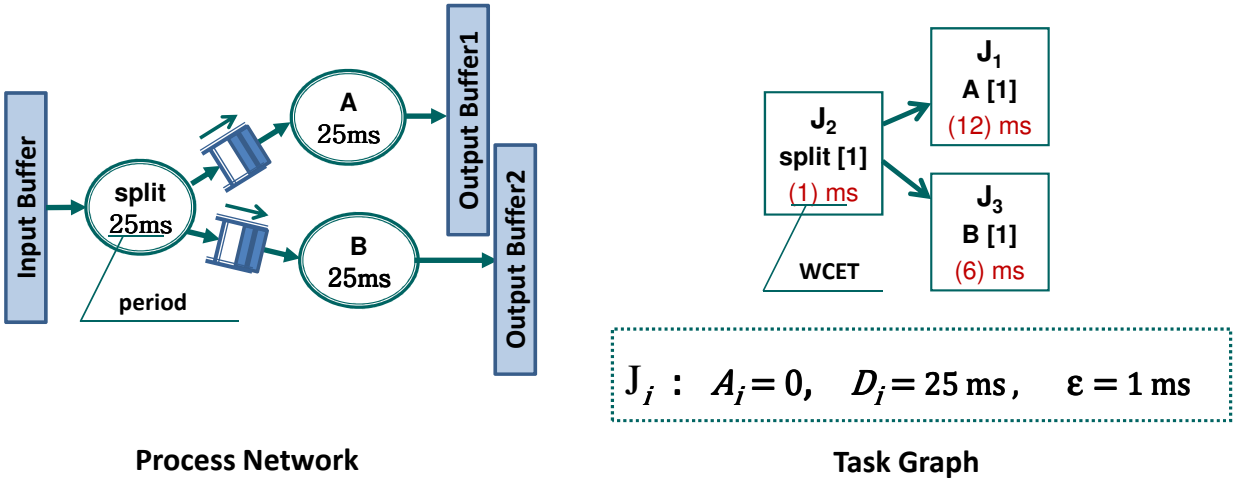


Fig. 8: The FPPN model of a system with three tasks

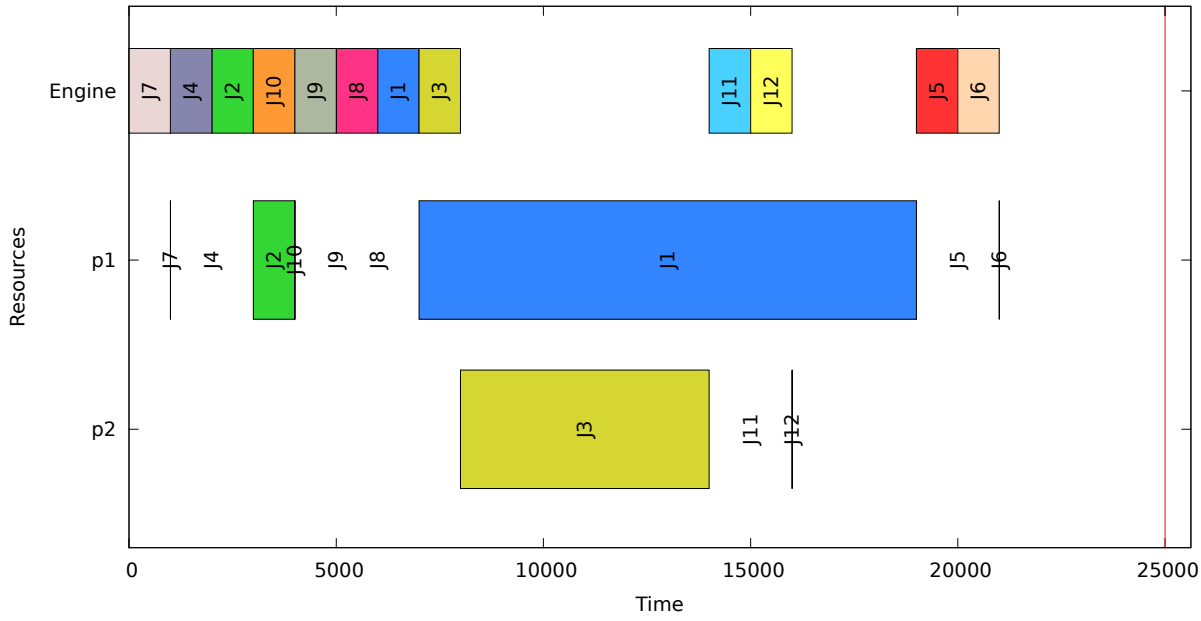


Fig. 9: FPPN schedule computed by the static scheduler for the three-tasks example

on the fixed priority policy simulation [38]. For the task graph of Figure 8 two CPU cores are needed. This happens due to the 12 ms interference overhead (four BIP interactions required per job execution take $4{\cdot}\epsilon = 4$ ms); the task graph cannot be scheduled on a single processor, because an amount $3 \cdot 4 \cdot \epsilon + (1+12+6)$ ms $= 31$ ms of processor time per period of 25 ms has to be allocated. In the schedule of Figure 9, Task 'Split' and Task 'A' are mapped to Core 1 and Task 'B' to Core 2.

The static scheduler produces the time-triggered table depicted in Figure 9, and the schedule also includes the virtual jobs that represent the execution of BIP RTE engine. By our convention, Core 0 is reserved for the engine, so that is where the virtual jobs are scheduled. From the schedule we see that the schedule intervals for the application jobs J1, J2 and J3 are preceded by gaps where the jobs wait until the engine handles the BIP transitions, executed as virtual jobs on Core 0. This phenomenon illustrates the idea on how we model and handle interference, for more details see [36].

As it was mentioned earlier, representing the schedule by BIP components and plugging them into the BIP model is not yet automated. Thus, this is done manually by inserting the additional $\mathcal{FP}$ components, which should impose the computed schedule on the system.

## 9 Case Study: a Spacecraft On-board Guidance, Navigation and Control System

The space industry aims to utilize state-of-the-art multi-core processors, which will allow to reduce size, weight, cost, and power consumption, while software predictability and safety in terms of timing and functional behavior should be preserved with a sufficient level of assurance. Our framework was employed within the context of a space-industry collaborative project, to port a Guidance Navigation & Control (GN&C) on-board production-grade application, originally conceived for a uni-processor, onto the quad-core symmetric multiprocessing LEON4FT architecture [2] based on the rad-hard chip NGMP (Next Generation Microprocessor) of the European Space Agency.

### 9.1 Application Design with the TASTE-to-BIP framework

The GN&C module is an on-board real-time software with hard deadlines that regulates the spacecraft movement by processing the sensor data and controlling the associated actuators. The GN&C application computes the orbital location required to satisfy the mission requirements, based on the guidance equipment. Subsequently, the navigation routines track the actual spacecraft location, while the flight controller relocates the orbital accordingly. The GN&C subsystem comprises the four tasks listed below.

(i) The *Guidance Navigation Task* that is responsible to execute the guidance and navigation algorithms by estimating the current translational state of the spacecraft, taking into account the actuators data and the sensors' measurements. This task also computes the derived air data parameters and aerodynamic, while keeping the vehicle on track during the flight to reach the desired location for parachute triggering. It calculates the actual location and provides the reference attitude and the calculated air data and aerodynamic parameters for the control task. This specific task is built as a periodic process with period $T_p = 500$ms, deadline $d_p = 500$ms and worst-case execution time $C$=22ms. Its criticality is at level D.

(ii) The *Control FM task* is also a periodic process with period $T_p = 50$ms, deadline $d_p = 50$ms, worst-case execution time $C$=8ms and criticality level D. This task executes the control and flight management algorithms.

(iii) The *Control Output Task* is a D-critical periodic process with period $T_p = 50$ms, deadline $d_p = 50$ms and worst-case execution time $C$=4ms. It is in charge of sending the outputs of the GN&C subsystem (i.e. mach, dynamic pressure, longitude and geodetic values) to the Dynamics Kinematics and Environment (DKE) module.

(iv) The *Data Input Dispatcher Task* is invoked each time new sensor data is available. This process reads, decodes, and dispatches the input data to the other tasks. The input data is categorized as MVM (Mission and Vehicle Management), IMU (Inertial Mea-

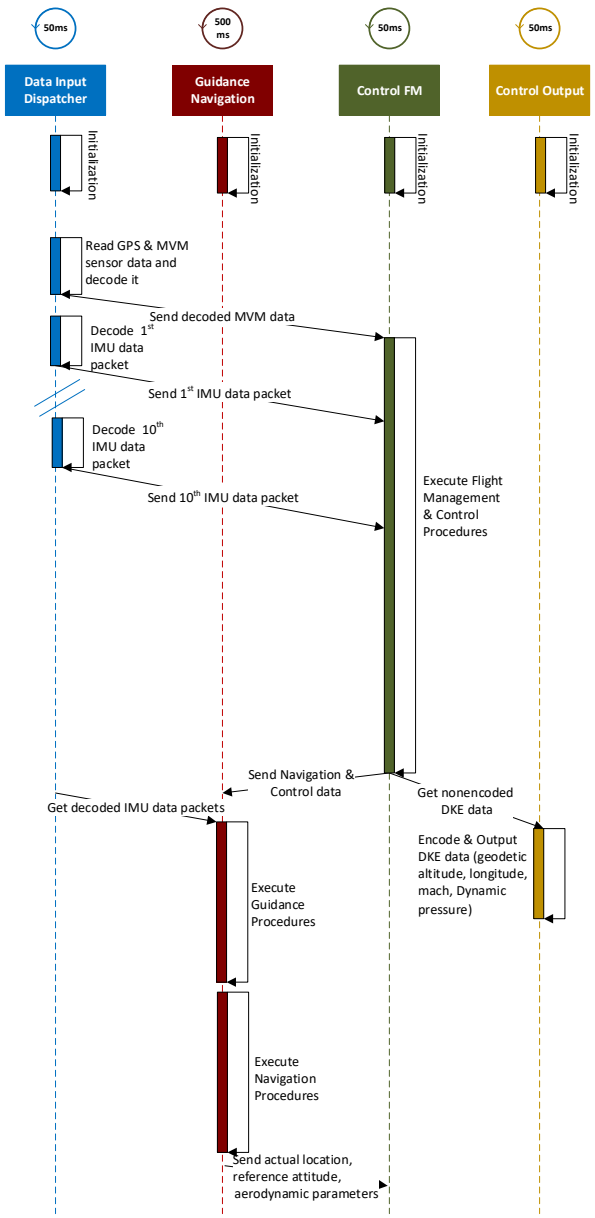


Fig. 10: The MSC of GN&C execution within a single Hyperperiod of 500ms

surement Unit) and GPS (Global Positioning System) data. For the purpose of this study, the external data was not communicated via the chip I/O channels, but had been measured and stored in advance as static data for the application executable. The IMU data processing is performed in 10 packets per global cycle of 500ms. This is a periodic process with period $T_p$ = 50ms, deadline $d_p$ = 50ms and worst-case execution time $C$=6ms. The task criticality is also at level D.

The data flow between the GN&C tasks within a hyperperiod of H = 500ms, after removing all RTEMS (Real-Time Executive for Multiprocessor Systems) calls and timers, is shown in the Message Sequence Chart (MSC) of Figure 10. The tasks' WCETs were estimated by profiling the execution of the application under the BIP RTE.

We considered two different implementation scenarios to highlight the applicability of our approach by exercising both our fully automated design flow, as well as a similar flow with user intervention at different design steps with the aim to refine the model behavior and improve the implementation efficiency. Specifically, we explored two versions of the GN&C application design. Scenario A was based on the timing constraints of the original specification, utilizing two of the four available LEON4FT CPU cores. In this case, the Control Output and the Guidance Navigation tasks exhibit respectively 30ms and 450ms time offsets (close to the end of their periods). Scenario B was a 'pipelining' scenario, in which we aimed to utilize three computing cores by assuming that there was headroom to increase the end-to-end latency constraint (i.e. the deadline) of the two aforementioned tasks to exceed their period. Within this context, in Scenario B the user modifies the generated BIP model and the task graph, so as to remove the offsets from the Control Output and Guidance Navigation tasks and to shift every job execution of those tasks into the next period. Consequently, having their input data prepared in the previous period,these tasks can start executing on different processor cores, in parallel with the two other tasks. Our intent for Scenario-B is to test an implementation where the system could process more data samples per unit of time, so that the application designer could reduce the task periods to improve the throughput and hence the quality of operation.

We first developed a TASTE architecture description for the GN&C FPPN model. At the same time, we had to adapt the functional code of tasks for use in our framework, by removing the low-level RTEMS calls for multi-tasking from the original mono-core implementation and adding high-level calls to TASTE interfaces for explicit data communication according to our MoC. We note that both scenarios use the same TASTE model structure that is shown in Figure 11.

The TASTE architectural model was fed into our TASTE-to-BIP framework and was automatically transformed into a BIP model, while the task graph for a hyperperiod was also generated. We then passed the task graph to the offline scheduling tool, which first calculated the system load[3], in order to determine the minimum required number of processor cores, while taking into account the precedence constraints between jobs [38] and the multi-core interference [36]). For Scenario A, the system load calculated by the tool was 112%, *i.e.,* at least two cores are required. For two cores, the offline scheduler generated a static schedule, demonstrating that all deadlines could be met, thus giving the verdict that the system is schedulable. We did not encounter any need to insert additional FP components for the online scheduling, because all tasks mapped by the scheduler to the same core were already fully ordered by the functional priority constraints of the application model.

The BIP model was then compiled and linked with the BIP RTE for the LEON4 multi-core processor by adopting a thread-to-CPU core affinity rule, which means that the BIP RTE performs one-to-one thread mapping to cores (therefore, the terms thread and core are used as synonyms). The executable was loaded and ran on the LEON4FT board and the task execution traces were collected and depicted in Gantt charts form. We note that the current version of BIP RTE utilizes an additional CPU core. Consequently, for Scenario A, one more core was required, which made three cores in total.

The two case study scenarios are presented in the next two subsections in more detail. For brevity, we often avoid namimg the 4 tasks and prefer using a short label 'P<id>' instead, where '<id>' is a numeric process identifier. *P1* stands for the Data Input Dispatcher, *P2* for the Control FM, *P3* for the Control Output and *P4* for the Guidance Navigation task. Since the BIP RTE occupies a separate core and has a certain overhead, we use *P20* for the 'job' of a 'virtual task' that represents handling a discrete BIP interaction by the RTE. One virtual job execution comprises the online checking of the run-time status of all BIP components, determining which new BIP automata transitions/interactions are ready for execution and executing one of them. We note that the BIP RTE executes only the *discrete* transitions, which in our FPPN model mainly correspond to 'Start', 'Finish', 'Deadline', 'Invoke' and 'FalseInvoke' transitions. Continuous-time transitions, including all computations of the tasks and their 'Read' and 'Write'

[3] In scheduling theory this metric is similar to processor utilization.

actions, are executed in parallel by the tasks on the different cores. The tasks synchronize with the BIP RTE only on their 'Start' and 'Finish' interactions, which are executed synchronously with the BIP engine.

### 9.2 Scenario-A

The functional priority indexes *FP* for the tasks in Scenario A were assigned as follows:

(i) Data Input Dispatcher: *FP=1*
(ii) Control FM Task: *FP=2*
(iii) Control Output Task: *FP=3*
(iv) Guidance Navigation Task: *FP=4*

This specific precedence order was chosen by analyzing the task execution order shown in the MSC diagram (Figure 10) and the application's source code, in particular how the threads of different tasks interact by conditional signals. Figure 12 depicts the GN&C FPPN model, where the functional priorities impose precedence from the numerically smaller *FP* indices (i.e., higher-priority) to the numerically larger ones. The length of the 'Navigation_DATA_MB' mailbox was set to 10, due to the fact the Guidance-Navigation task requires 10 IMU packets to be processed before it can execute, which translates to an offset of 9 periods of the other tasks (i.e. 450 ms). Figure 13 delineates the task graph generated by the TASTE-to-BIP tool for a hyperperiod of 500ms.

We executed Scenario A on the LEON4FT CPU and observed the four discrete interactions per 1 job execution and 31 jobs per hyperperiod (specifically 31×4=124 discrete transitions are executed by the BIP RTE per hyperperiod as it is shown in the measured Gantt chart of Figure 14 for a hyperperiod 500ms plus a 100ms timing-window). The P20 activities were mapped to Core 0, whereas the jobs of tasks P1, P2 and P3 were mapped to Core 1 and those of P4 to Core 2. The job of P4 is executed right after 10 consecutive jobs of P1 and P2 and 9 jobs of P3. This job is delayed due to the 450ms invocation offset and its least functional priority. Since P3 and P4 do not communicate via the channels ($(P3, P4) \notin \mathcal{FP}$) it is possible to execute them in parallel, which was actually programmed in our static schedule. In fact, this was necessary in order to satisfy the deadlines, because , as mentioned earlier, the system load exceeded 100%.

In order to explore existing possibilities for reducing the encountered execution overhead, let us recall that there are 124 transitions executed by the BIP engine on Core 0 per one hyperperiod. In an optimized BIP model, it would be desirable to share Arrival and Deadline transitions between the tasks, which would cut the P20 overhead by almost 50% (a scenario to be explored in future work). To further reduce the overhead, we could possibly merge the tasks with short execution times, such as P1 and P2, into one.

### 9.3 Scenario-B

In the pipelined version of the GN&C application, we aim to increase the number of LEON4FT cores utilized in parallel, in order to evaluate the possibility of increasing the throughput [23]. To this end, we modified the execution order of the tasks in a way that does not violate their data dependencies. This modification was accomplished by changing the functional priority, though when doing this the user always has to take into account the tasks' dependencies in the application. This change is performed through the TASTE architecture description and is an input into the TASTE-to-BIP framework. The user will also have to intervene and change the functional priorities during the late design steps, if he realizes that the real-time constraints are not met. In our case, by changing the functional priority relation of the Control Output (P3) and Guidance Navigation (P4) tasks, it was possible to process the data received in the previous period, while the rest of the tasks operate in current period. Effectively, the offsets were prolonged from 30ms to 50ms for the Control Output Task and from 450ms to 500ms for the Guidance Navigation Task. This meant that P3 and P4 tasks could be executed at the start of every next period, since the necessary data to operate was available at this timing point. To preserve the functional behavior in Scenario B, these tasks should execute earlier, so we assigned them higher functional priorities, *i.e.,* smaller FP indices.

Furthermore, to ensure the extra parallelism, a buffering scheme was incorporated by increasing the mailbox (FIFO) size between Data Dispatcher and Guidance Navigation tasks by one position (11 memory addresses in total), so that they could run in parallel and their relative functional priorities will not affect them (therefore, the mailbox channel has no functional priority arrow associated with it). As was previously mentioned, priority arrows exclude data races and protect the data channel interfaces. In our case, the use of a double buffer ensured that P4 (reader) could read one data value from mailbox, while P1 (writer) could concurrently write another data value to the mailbox. With this modification in the FPPN model, we allowed P1, P3 and P4 to execute in parallel, since they had no predecessors. The derived FPPN model for Scenario B is shown in Figure 15, where the *FP* indices were assigned as follows:

(i) Data Input Dispatcher: *FP=1*
(ii) Control Output Task: *FP=2*
(iii) Guidance Navigation Task: *FP=3*

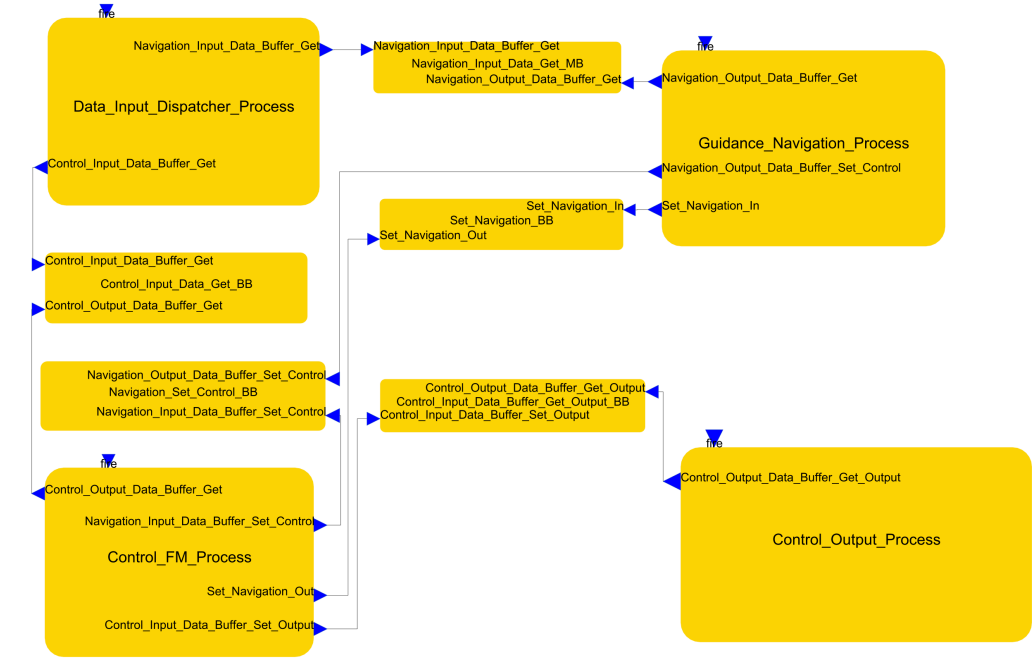


Fig. 11: The GN&C application in TASTE (both implementation scenarios have this structure)

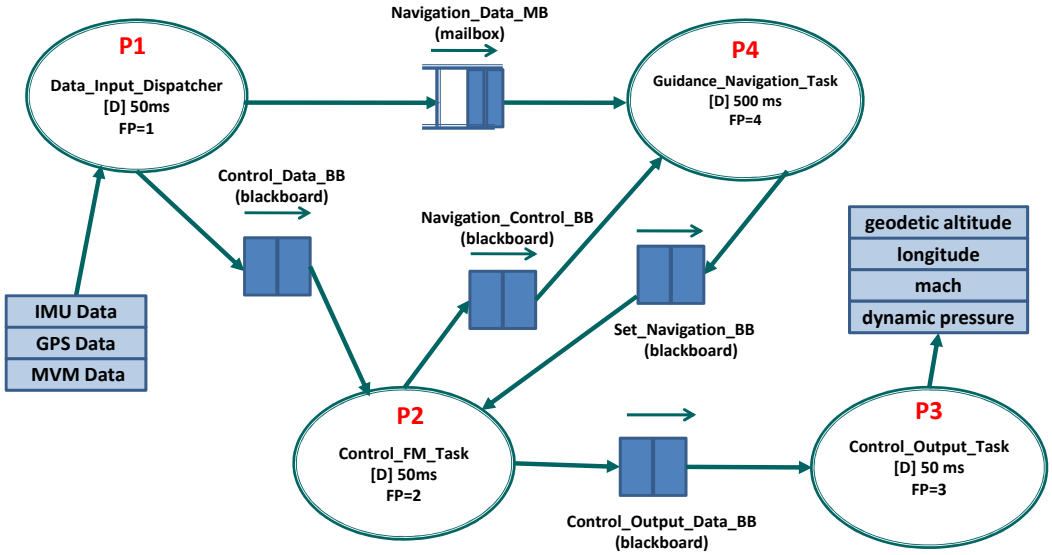


Fig. 12: The GN&C FPPN model (Scenario-A)

(iv) Control FM Task: *FP=4*

The generated task graph in Figure 16 yielded a system load 65%, which is significantly smaller than in Scenario-A, due to removal of $\mathcal{FP}$ from the FPPN model, which resulted in fewer precedence constraints. Figure 17 illustrates the measurements obtained during the execution of Scenario B on the LEON4FT CPU. All the four CPU cores were utilized and the deadline constraints were met. Specifically, P3 was executed on Core-3 and in parallel, P1 and P4 were executed respectively on Core 1 and Core 2, while the BIP RTE engine ran the BIP transitions on the fourth core (Core 0). Although a higher parallelism was achieved in comparison to Scenario A, the throughput was not increased due to the interference between P1 and P4, which resulted in longer execution time for P4. We observed that it took 38ms after the next period start, for the jobs of P3 and P4 to complete their execution.

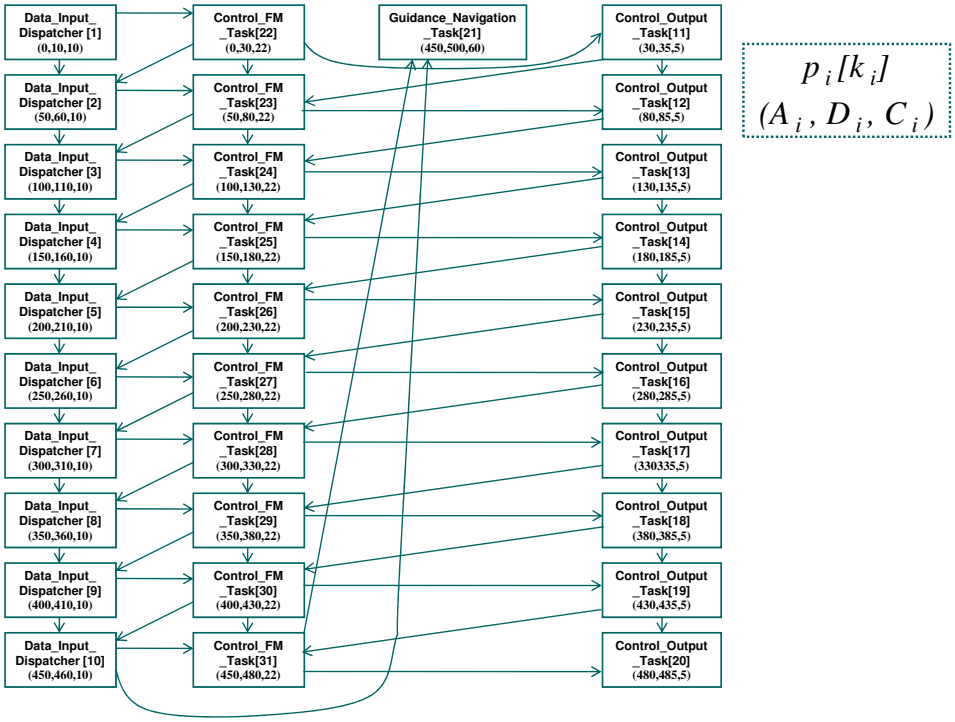


Fig. 13: Task graph for Scenario-A

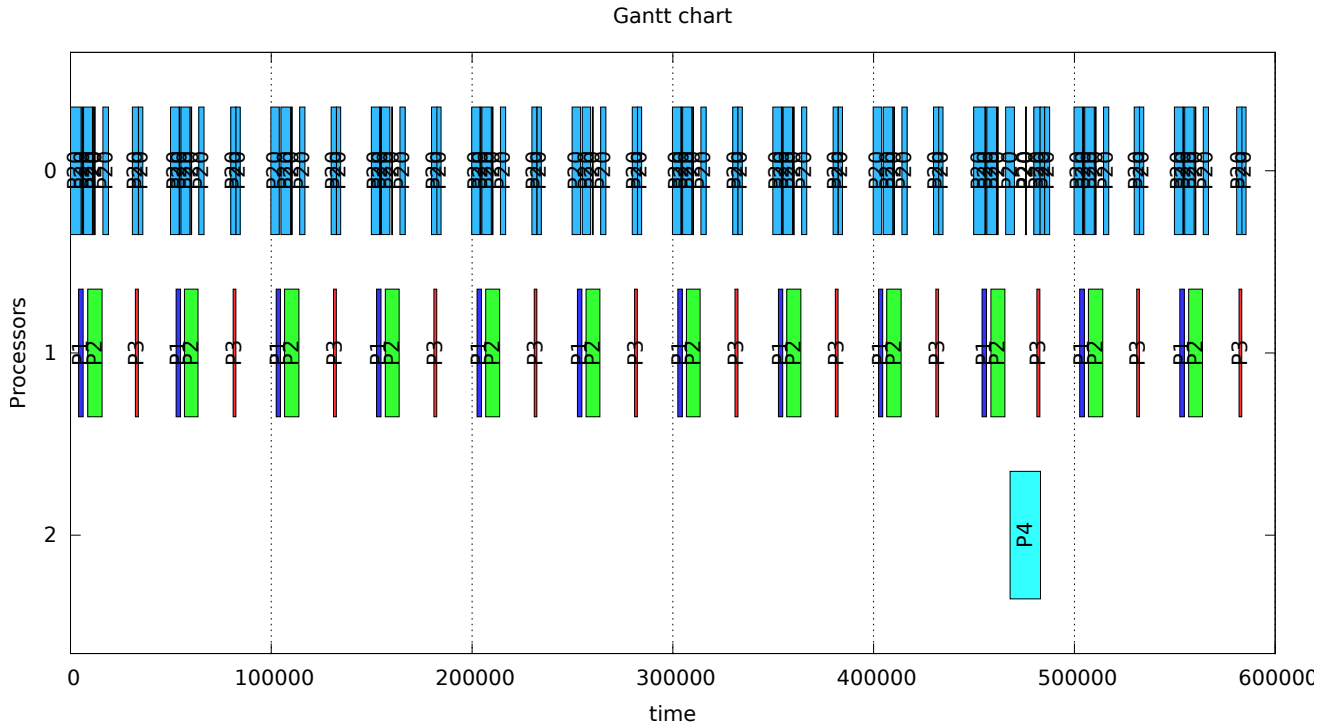


Fig. 14: Execution of the GNC application on LEON4FT (in microseconds)

### 9.4 Discussion

Through the GN&C case study we had the chance to evaluate our model-based design flow that utilizes the FPPN MoC. The implementation of Scenario A resulted in a straightforward application of the model's seman-

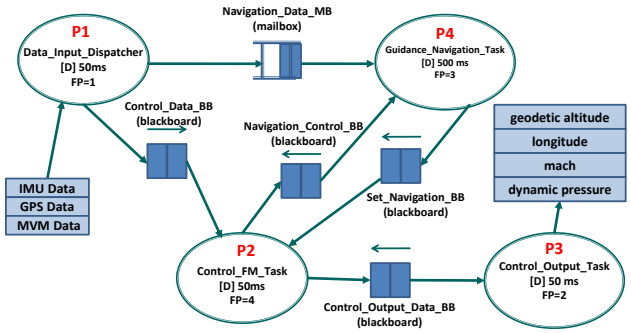


Fig. 15: The GN&C FPPN model (Scenario-B)

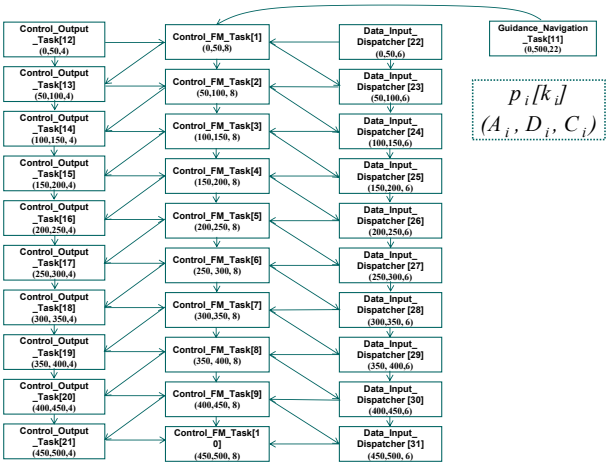


Fig. 16: Task graph for Scenario-B

tics and its associated design flow. On the other hand, it was also shown how the designer could intervene in specific design steps towards modifying the implementation, while preserving the FPPN semantics. Specifically, since the TASTE-to-BIP tool is not aware of the extra buffering place employed in Scenario B in the mailbox channel, we had to modify the generated task graph by deleting the two job-precedence arrows (J22 $\rightarrow$ J11 and J11 $\rightarrow$ J23) associated with the precedence constraints between the Data Input Dispatcher and Guidance Navigation tasks (note the absence of priority arrow on top of mailbox). The resulted task graph is the one shown in Figure 16. Additionally, we delayed the Control-Output task such that to deliver its output with a 20ms a delay (i.e. the time-offset for Control-Output task is 50ms in Scenario B instead of the 30ms offset in Scenario A), which was tolerable to sustain the correct operation of the GN&C subsystem.

Recall that in our BIP model all jobs are synchronized with the BIP RTE engine at the 'Start' and 'Stop' interactions, whereas the engine is busy executing 124 interactions per 500ms with each one taking 1ms, which means that the BIP engine is busy for 25% of the time. Thus, when a task component tries to execute a BIP interaction via a BIP engine, the engine may turn out to be unresponsive for some time (*e.g.*, busy with other tasks), so the given task will have to wait for its turn. This causes some minor idle-processor gaps between the task execution intervals and can be interpreted as interference between the tasks for accessing the BIP engine. Similar forms of inter-task interference exists in all multi-tasking real-time environments, while it is modeled and taken into account by our scheduling tools [36], when estimating the system load and applying our static scheduling tool to obtain from it a 'schedulability' verdict.

In Scenario B, where more tasks execute in parallel on different cores, we noticed additional interference, which were reflected to the tasks by requiring more time to complete their jobs than the estimated WCET. Although the pipelined version of Scenario B utilizes more cores than Scenario A, we noticed that the parallelism gain did not effectively speed-up the application on LEON4FT. This happens mainly due to the additional interference between P1 and P4, which extends the execution times of both beyond the WCET's respected in Scenario A. We suspect that this interference arises due to the imperfect implementation of synchronization of BIP interactions between tasks, when two tasks access the same channel, since in particular P1 and P4 execute in parallel and access the mailbox. The nature of this interference is under study towards improving the BIP RTE implementation, so as to reduce it. If this is not possible, the scheduling models will have to be updated, in order to take into account this interference.

## 10 Conclusions

We presented a model-based design approach, which allows deriving correct-by-construction multi-core implementations of reactive streaming software, when the software is programmed using the high-level FPPN model of computation. Applications are programmed independently from the execution platform, while sustaining determinism in task parallelism by construction. At the same time, through a gradual refinement of the system's design it is possible to configure real-time attributes that ensure schedulability and a predictable timing behavior. The design framework allows to explore various implementation scenarios, in order to improve the resource utilization, without invalidating the guarantees offered by the FPPN model of computation. The means to achieve all these is a careful definition of the FPPN semantics at two distinct levels, while preserving fundamental correctness properties, and a model

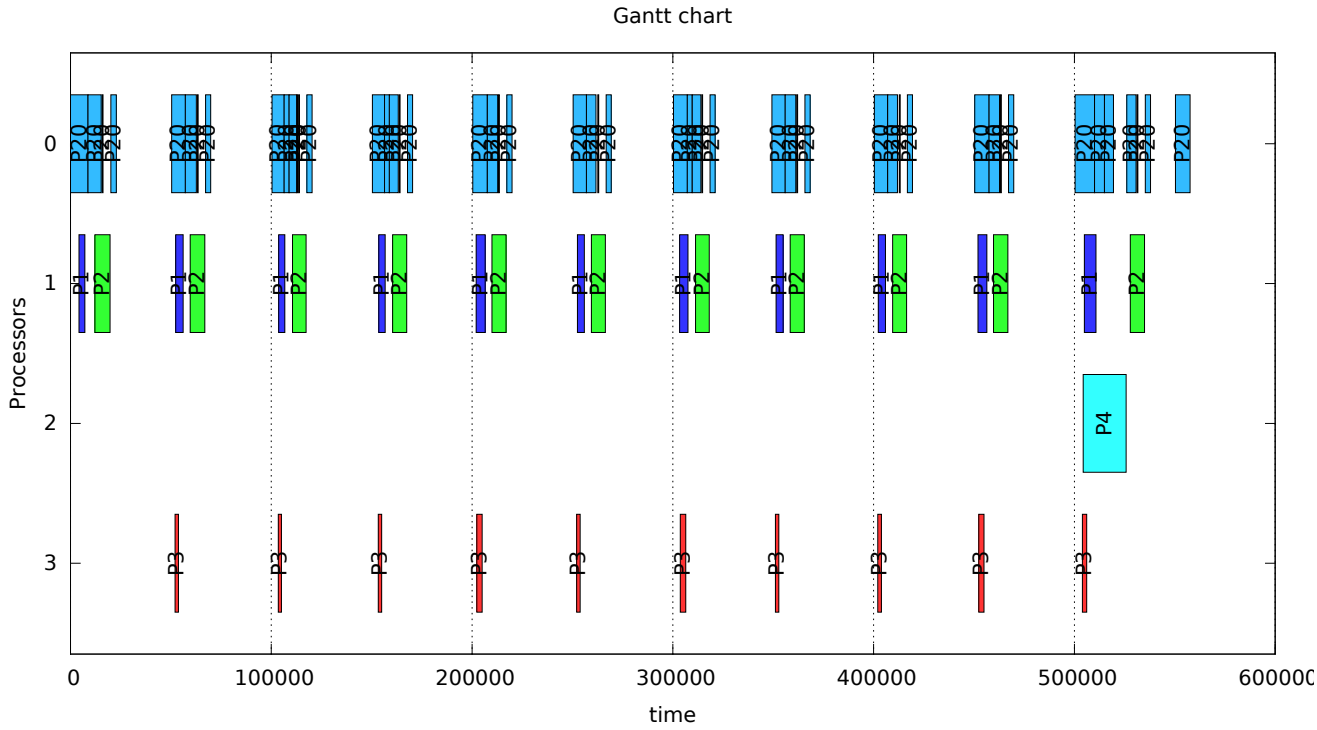


Fig. 17: Real-Time execution of Scenario-B on LEON4FT (in microseconds)

transformation framework for transcribing FPPNs into an executable formal specification language called BIP, for modeling connected timed automata. The designer can specify FPPNs by embedding functional code into a high-level description of the software architecture using the TASTE toolset.

The proposed method was evaluated through the implementation of a real-time application for satellites on ESA's next generation multi-core processor. Within this context two implementation scenarios were studied.

Compilation from a model of computation to an executable model based on automata is not new [21]; a contribution of this paper is to show how this is done for the FPPN model. Such an approach has the advantage of obtaining an executable model which, being automata based, is potentially easier to verify than low-level middleware code. On the other hand, such an approach raises the problem of certification of the code generator that translates the automata-based language into the binary code and the problem of certifying the associated run-time engine. Another topic of research is overhead minimization, as it was shown in our case study, where we had the chance to explore a multitude of possibilities for substantial manual improvements of the generated code. These issues are subjects of future research prospects.

Additional future work also includes the design framework's improvement, so as to automate the design step involving the offline and online schedulers, and to enhance the interference awareness of the scheduling methods. Also, statistical tools as described in [37] are to be incorporated towards deriving reliable WCET estimations. Finally, the possibility to include the ITU-T SDL language or Simulink at the architectural level of our framework is also considered.

## References

1. Autofocus tool. URL `https://af3.fortiss.org/`. [Available Online; accessed: 05-February-2019]
2. GR-CPCI-LEON4-N2X: Quad-core LEON4 next generation microprocessor evaluation board, http://www.gaisler.com/index.php/products/boards/gr-cpci-leon4-n2x
3. Time-critical applications on multicore platforms. URL `http://www-verimag.imag.fr/Time-Critical-Applications-on-Multicore.html`
4. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: Proceedings of the Tenth ACM International Conference on Embedded Software, EMSOFT '10, pp. 229–238. ACM, New York, NY, USA (2010). DOI 10.1145/1879021.1879052. URL `http://doi.acm.org/10.1145/1879021.1879052`
5. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times: A tool for schedulability analysis and code generation of real-time systems. In: K.G. Larsen,

P. Niebert (eds.) Formal Modeling and Analysis of Timed Systems, pp. 60–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
6. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
7. Basu, A., Bensalem, S., Bozga, M., Bourgos, P., Maheshwari, M., Sifakis, J.: Component assemblies in the context of manycore. In: B. Beckert, F. Damiani, F.S. de Boer, M.M. Bonsangue (eds.) Formal Methods for Components and Objects: 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers, pp. 314–333. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Brau, G., Hugues, J., Navet, N.: Towards the systematic analysis of non-functional properties in model-based engineering for real-time embedded systems. Science of Computer Programming **156**, 1 – 20 (2018). DOI https://doi.org/10.1016/j.scico.2017.12.007. URL http://www.sciencedirect.com/science/article/pii/S0167642317302927
9. Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T., Weber, R.: The design of distributed systems - an introduction to focus. Tech. Rep. TUM-I 9202-2, Technische Universitat Munchen (1992)
10. Broy, M., Fox, J., Hölzl, F., Koss, D., Kuhrmann, M., Meisinger, M., Penzenstadler, B., Rittmann, S., Schätz, B., Spichkova, M., Wild, D.: Service-Oriented Modeling of CoCoME with Focus and AutoFocus, pp. 177–206. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). DOI 10.1007/978-3-540-85289-6_8. URL https://doi.org/10.1007/978-3-540-85289-6_8
11. Broy, M., Stolen, K.: Systems: Focus on Streams, Interfaces, and Refinement, p. 348. Springer-Verlag New York (2001). DOI 10.1007/978-1-4613-0091-5
12. Chaki, S., Kyle, D.: DMPL: Programming and verifying distributed mixed-synchrony and mixed-critical software. Tech. rep., Carnegie Mellon University (2016). URL http://www.andrew.cmu.edu/user/schaki/misc/dmpl-extended.pdf
13. Claraz, D., Grimal, F., Laydier, T., Mader, R., Wirrer, G.: Introducing multi-core at automotive engine systems. In: ERTSS'14, Embedded Real-time Software and Systems (2014)
14. Cordovilla, M., Boniol, F., Forget, J., Noulard, E., Pagetti, C.: Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In: RTNS (2011)
15. 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. Proceedings of the IEEE **91**(1), 127–144 (2003)
16. Feiler, P., Gluch, D., Hudak, J.: The architecture analysis & design language (AADL): An introduction. Tech. Rep. CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2006). URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879
17. Forget, J., Boniol, F., Grolleau, E., Lesens, D., Pagetti, C.: Scheduling dependent periodic tasks without synchronization mechanisms. In: RTAS'10, pp. 301–310
18. Fuhrmann, H., von Hanxleden, R., Rennhack, J., Koch, J.: Model-based system design of time-triggered architectures - avionics case study. In: 2006 ieee/aiaa 25th Digital Avionics Systems Conference, pp. 1–12 (2006). DOI 10.1109/DASC.2006.313745
19. Geilen, M., Basten, T.: Reactive process networks. In: EMSOFT'04, pp. 137–146. ACM (2004)
20. Ghamarian, A.H.: Timing analysis of synchronous dataflow graphs. Ph.D. thesis, Eindhoven University of Technology (2008)
21. Giannopoulou, G., Poplavko, P., Socci, D., Huang, P., Stoimenov, N., Bourgos, P., Thiele, L., Bozga, M., Bensalem, S., Girbal, S., Faugere, M., Soulat, R., de Dinechin, B.D.: DOL-BIP-Critical: a tool chain for rigorous design and implementation of mixed-criticality multi-core systems. Design Automation for Embedded Systems **22**(1), 141–181 (2018). DOI 10.1007/s10617-018-9206-3. URL https://doi.org/10.1007/s10617-018-9206-3
22. Gioulekas, F., Poplavko, P., Katsaros, P., Bensalem, S., Palomo, P.: A process network model for reactive streaming software with deterministic task parallelism. In: A. Russo, A. Schürr (eds.) Fundamental Approaches to Software Engineering, pp. 94–110. Springer International Publishing, Cham (2018). DOI 10.1007/978-3-319-89363-1_6
23. Gioulekas, F., Poplavko, P., Katsaros, P., Palomo, P.: Process network models for embedded system design based on the real-time bip execution engine. In: S. Bliudze, S. Bensalem (eds.) Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, Thessaloniki, Greece, 15th April 2018, *Electronic Proceedings in Theoretical Computer Science*, vol. 272, pp. 79–92. Open Publishing Association (2018). DOI 10.4204/EPTCS.272.7
24. 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)
25. Halbwachs, N.: Synchronous Programming of Reactive Systems. Springer-Verlag, Berlin, Heidelberg (2010)
26. Hansson, A., Goossens, K., Bekooij, M., Huisken, J.: CoMPSoC: A template for composable and predictable multi-processor system on chips. ACM Transactions on Design Automation of Electronic Systems (TODAES) **14**(1), 2 (2009)
27. Hölzl, F., Feilkas, M.: 13 AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems, pp. 317–322. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-16277-0_13. URL https://doi.org/10.1007/978-3-642-16277-0_13
28. Huber, F., Molterer, S., Rausch, A., Schatz, B., Sihling, M., Slotosch, O.: Tool supported specification and simulation of distributed systems. In: In International Symposium on Software Engineering for Parallel and Distributed Systems, pp. 155–164. B. Kramer, N. Uchihira, P. Croll, and S. Russo Eds., IEEE (1998)
29. 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)
30. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**(1), 1–34 (2004)
31. Kahn, G.: The semantics of a simple language for parallel programming. In: J.L. Rosenfeld (ed.) Information Processing '74: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York, NY (1974)
32. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Transactions on Computers **C-36**(1), 24–35 (1987)
33. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987). DOI 10.1109/PROC.1987.13876

34. Mkaouar, H., Zalila, B., Hugues, J., Jmaiel, M.: From aadl model to lnt specification. In: Reliable Software Technologies Ada-Europe 2015, Springer International Publishing, pp. 11–20. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-19584-1_10
35. Perrotin, M., Conquet, E., Delange, J., Schiele, A., Tsiodras, T.: TASTE: A real-time software engineering tool-chain overview, status, and future. In: I. Ober, I. Ober (eds.) SDL 2011: Integrating System and Software Modeling. Int. SDL Forum. Revised Papers, pp. 26–37. Springer, Berlin, Heidelberg (2012)
36. Poplavko, P., Kahil, R., Socci, D., Bensalem, S., Bozga, M.: Mixed-critical systems design with coarse-grained multi-core interference. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques, ISoLA'16, pp. 605–621. Springer International Publishing (2016). DOI https://doi.org/10.1007/978-3-319-47166-2\textunderscore42
37. Poplavko, P., Nouri, A., Angelis, L., Zerzelidis, A., Bensalem, S., Katsaros, P.: Regression-based statistical bounds on software execution time. In: Verification and Evaluation of Computer and Communication Systems - 11th International Conference, VECoS 2017, Montreal, QC, Canada, August 24-25, 2017, Proceedings, pp. 48–63 (2017). DOI https://doi.org/10.1007/978-3-319-66176-6\textunderscore4
38. Poplavko, P., Socci, D., Bourgos, P., Bensalem, S., Bozga, M.: Models for deterministic execution of real-time multiprocessor applications. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE '15, pp. 1665–1670. EDA Consortium, San Jose, CA, USA (2015). URL http://dl.acm.org/citation.cfm?id=2757012.2757198
39. Saidi, S.: On the benefits of multicores for real-time systems. In: 2017 Euromicro Conference on Digital System Design (DSD), pp. 383–389 (2017). DOI 10.1109/DSD.2017.85
40. Socci, D., Poplavko, P., Bensalem, S., Bozga, M.: A timed-automata based middleware for time-critical multicore applications. In: 2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, pp. 1–8 (2015). DOI 10.1109/ISORCW.2015.55
41. Triki, A., Combaz, J., Bensalem, S., Sifakis, J.: Model-based implementation of parallel real-time systems. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13, pp. 235–249. Springer-Verlag, Berlin, Heidelberg (2013)
42. Waez, M.T.B., Dingel, J., Rudie, K.: A survey of timed automata for the development of real-time systems. Computer Science Review **9**, 1–26 (2013)
43. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From aadl to timed abstract state machines: A verified model transformation. Journal of Systems and Software **93**, 42 – 68 (2014). DOI https://doi.org/10.1016/j.jss.2014.02.058. URL http://www.sciencedirect.com/science/article/pii/S0164121214000727
44. Yang, Z., Hu, K., Ma, D., Pi, L.: Towards a formal semantics for the aadl behavior annex. In: 2009 Design, Automation Test in Europe Conference Exhibition, pp. 1166–1171 (2009). DOI 10.1109/DATE.2009.5090839