Taste2BIP User Manual

Technical Note

Fotios Gioulekas, CERTH

Peter Poplavko, Verimag

Alexandros Zerzelidis, CERTH

Panagiotis Katsaros, CERTH

December 2017

Abstract

In this document, we give a brief overview of Taste2BIP, a design-flow and code generation prototype tools developed in the MoSATT-CMP activity, a two-year collaborative project 2014-2016. This user-guide describes how a real-time application which targets its operation onto embedded multi-core platforms can be developed as a Fixed Priority Process Network model and automatically deployed on the platform. As a platform example we mainly use Next Generation Microprocessor (NGMP) platform with the LEON4 multicore processor. Fixed Priority Process Network is a consolidated Model of Computation using an amendment TASTE tool, which is an open-source real-time software engineering environment openly available on a virtual machine. For deployment, the TASTE model is first translated into equivalent DOL Critical, which is an academic model conceptually comparable to TASTE, developed at ETH Zurich. For deployment it is translated into real-time BIP timed automata model, corresponding to a network of communicating timed automata that can execute C++ code at transitions. Using real-time BIP real-time environment (RTE), the design can be simulated or deployed on a multi-core platform.

Table of Contents

Index of Figures

CHAPTER 1. Model-Based Tool Flow for Multi-Core Applications

1.1 Introduction

Our design flow is based on the “Fixed Priority Process Networks” (FPPN) model of computation (MoC). It supports seamless model transformations and incorporates as front-end the open source TASTE tool, a TASTE 2 BIP tool chain, and the BIP tools. Figure 1 depicts the MoSATT-CMP design flow. TASTE is a robust and open-source tool-chain for software development, which targets heterogeneous embedded systems using a model-centric development approach. At the system-level part of the design flow, the FPPN model of the real-time embedded SW application is conceptualized and modelled in TASTE-IV (TASTE Interface-View) editor, a graphical tool that aims at describing the logical interactions between the various functions of the system, which has been extended to support the FPPN MoC semantics, first of all by adding functional priorities for precedence between the tasks .The TASTE-IV editor generates AADL description of the graphical representation. The designer also defines the data-types exchanged via logical interactions and FPPN channels using the TASTE Data-View editor. In this editor, the ASN.1 language format is used for the definition of the data types. The C code skeletons from the TASTE IV model are generated together with its XML file format. The next step at the system-level design modelling is to proceed with the development of the functional code of the application, which in our design flow are based on the C language and calls to C++ libraries. The designer utilizes the TASTE Deployment-View (TASTE-DV) tool towards the successful compilation, binding and execution of the functional code of the FPPN model on the workstation platform. Within this context, just using the default TASTE tool suite, a first evaluation of the functional behaviour of the system under development can be done, though without taking into account the amendments added to TASTE, such as “functional priority”.

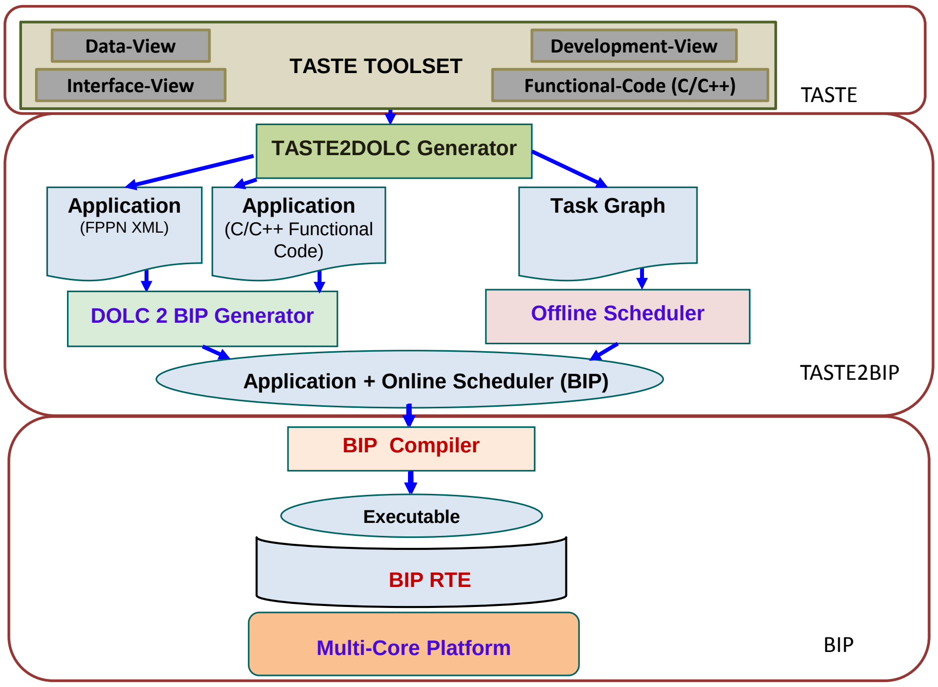


Figure 1: MoSATT-CMP Design Flow

At the first step, TASTE2 DOLC tool translates the application FPPN model and its functional code from TASTE IV into an academic design description language DOLC. A DOLC model includes both the task communication structure (here: the DOL XML) and the task functions (here: the C files and C/C++ libs). As the design evolves, the engineer invokes the DOLC 2 BIP generator, which is a tool that performs the transformation of the “descriptive” FPPN model in DOLC to a “behavioural” FPPN model in BIP. Note that both DOLC and BIP are extensions of previously existing frameworks to support the multi-core applications with hard mixed-critical real-time constraints [DOLC-BIP-Theory].

Note that currently support of ASN-generated code C data types is not yet automated, and instead we "fake" the data types by manually defined types in "temp_asn1.h" file included in process functional code. Instead real ASN-generated types could be used by including appropriated ASN-generated data types.

The inputs to the TASTE 2 BIP tool suite are the TASTE IV XML description of the FPPN model and C functional code. The design flow starts by TASTE 2 DOLC generator tool and then the flow is split into two sub-flows: the application sub-flow and the scheduler sub-flow. For application sub-flow TASTE 2 DOLC produces (transparently to the designer) the FPPN model in DOLC format (XML description and C code) which is then translated to BIP by DOLC2BIP generator, which translates the high-level models into equivalent BIP model. The BIP models define the executable semantics for all elements in the system, including the tasks and the schedulers. The BIP models can be both analysed and executed, e.g. for simulation purposes. We use two versions of BIP:

(a) one for single-thread simulation ("rtbip") with simulated time and zero task delay
(b) the other one for multi-threaded real-time execution ("multi-bip")

For the scheduler sub-flow, TASTE 2 DOLCderives from TASTE-IV specification of the FPPN MoC the corresponding static-scheduling problem instance (unrolled hyper-period). The problem instance specifies the jobs executed by each FPPN process within the time-frame of least-common multiple of the periods of all processes (termed as hyper-period). In the static schedule sporadic tasks are over-approximated by periodic ones. The problem instance in fact constitutes a task-graph, as it defines precedence arcs between the jobs based on their functional priorities, inter-process communication, and job arrival times (i.e., release time relative to hyper-period). Individual job properties are arrival time, absolute deadline, criticality level, and WCET values. Five levels of criticality are supported to distinguish between non-critical, mission-critical and safety-critical criticality levels i.e. A, B, C, D, and E. The scheduler sub-flow branches back into the application sub-flow to form one BIP model as the output.

It should be noted that integration of our mixed-critical scheduler and BIP RTE is still work-in-progress; in the currently RefSchedered tools we give a scheduler prototype that generates static schedule in textual form and in the form of graphical Gantt charts. In the generated BIP model instead we assume ASAP (self-timed) scheduling policy with static mapping of processes to cores, where the order of execution of processes is governed entirely by precedence constraints defined in DOLC XML format. By default these constraints are generated only for pairs of communicating processes to avoid data races. The user can manually add additional DOLC-XML "<precedence_chain>" relation for pairs of processes to avoid non-deterministic relative order of execution of two non-communicating processes mapped to the same core.

The bottom part of the figure represents the code generation from BIP to multicore platform. Currently we implicitly assume a shared memory multicore platform where tasks can be implemented using POSIX threads. The BIP components are translated to C++ classes that are orchestrated by multi-thread BIP runtime library (BIP-RTE is an extension of the BIP component-based design language to continuous time model closely related to timed automata). Note that the C++ language was used to facilitate the easy structuring of code together with data; we do not have a significant reliance on dynamic memory allocation, typical for some C++ applications. A special feature of the BIP framework is the co-programming of a scheduling policy together with application.

They are expressed as a network of interacting timed automata. This approach gives a two-sided benefit. On the one hand, it provides a rigorous formal semantics of system evolution in physical time, which facilitates formal analysis and proofs of system schedulability. On the other hand, BIP serves as a programming language with code generation and runtime environment (BIP RTE), which allows executing the BIP directly on the embedded platform. This realizes a principle "what you verify is what you execute", which facilitates the correct-by-construction argument of system design. Therefore, the model is deployed on the platform on top of the RT-BIP runtime environment (RTE) for multi-cores (hardware design level). The software model can also be combined with the hardware model to represent the complete software-hardware system and to perform timing analysis, WCET measurements for validation of schedulability properties.

1.1 Required External Packages

The following package needs to be installed (using sudo apt-get install)

- JRE (Java Runtime Environment (JRE)) v1.7 or higher

Required for model transformation tools: dol2bip and bipc.

- gnuplot

used to generate Gantt charts in scheduler tool and from traces generated by embedded multi-core platform (if available)

Optional:

- compilation toolchains for the respective commercial embedded multi-core platforms (with SMP enabled – if the platform uses RTEMS).

1.2 Installation of tools

```
> ls
installation   (dir)
exe            (dir)
examples       (dir)
instal.sh
```

Installation is simple, you just run "instal.sh" command in the root folder, this generates "sourceme.rc" file that should be sourced every time when you start using the tool chain in a new shell.

Some part of installation should be done manually.

Examine the installation folder. It contains two TASTE configuration files.

Extended ***TASTE_IV_Properties.aadl*** file

```
-- mapping TASTE : AADL
-- Container : Package
-- Function : System
-- Interface : Subprogram
property set TASTE_IV_Properties is
--  MyBoolean: aadlboolean applies to (System, Package);
--  MyString: aadlstring applies to (System, Subprogram);
   Version: aadlstring applies to (System);
--  MyInt: aadlinteger applies to (System);
--  MyReal: aadlreal applies to (System);
--  MyEnum: enumeration  (val1, val2, val3, val4) applies to (System);
   FPPNClass: enumeration (sporadic_process_A, sporadic_process_B, sporadic_process_C, sporadic_process_D,
sporadic_process_E, sporadic_protocol,periodic_process_A, periodic_process_B, periodic_process_C, periodic_process_D,
periodic_process_E, blackboard, mailbox) applies to (System);
   Fpriority: aadlinteger applies to (System);
   DataChannelSize: aadlinteger applies to (System);
   DataChannelLength: aadlinteger applies to (System);
end TASTE_IV_Properties;
```

The lines shown in bold are new lines. Add them into the "*TASTE_IV_Properties.aadl"* file at the following folder:

- */home/assert/tool-inst/share/config_ellidiss/*

Then, for this change to take effect immediately, manually copy the resulting file to the folder below

- */opt/Ellidiss-TASTE-linux/config/*

Please note! Whenever you run Update Taste script the "TASTE_IV_Properties.aadl" **is reset to default file** and then you need to repeat the modifications of this file again.

More information on TASTE *model extensions is available at:*

http://taste.tuxfamily.org/wiki/index.php?title=Technical_topic:_Extend_your_models_with_your_own_property_sets_to_hook_your_own_tools

CHAPTER 2. FPPN Modelling in TASTE-IV

In this chapter we provide guidelines for functional application modelling on TASTE-IV tool based on the FPPN semantics. A tutorial example which includes all the basic functionalities of an FPPN network is given.

2.1 Basics of FPPN modelling in TASTE

The TASTE-IV's function attributes **FPPNClass**, **Fpriority**, **DataChannelSize** and **DataChannelLength** have been added to TASTE-IV to model FPPN.

The ***FPPNClass*** attribute defines the type of the process node e.g. sporadic process, periodic process, sporadic-protocol and the channel type. It also determines the criticality level (i.e. A, B, C, D, and E) in case of sporadic or periodic process configuration.

The ***Fpriority*** is an integer value which dictates the “functional priority” number of the given process, and hence the priority orders in the network. Note that the concept of functional priority is different from “priority” in the usual meaning, i.e., it is not “scheduler priority”. The functional priority determines the order of simultaneous access to data channels by writer and reader process. Each process has always a unique functional priority integer number. We define that the process with the larger assigned integer value of *Fpriority* has lower priority. When reader and writer jobs are released at the same time then the lower priority job waits until the higher priority jobs finishes. This is so even in multi-core system, where, unlike scheduler priority, functional priority ensures mutual exclusion of writer and reader jobs in the priority-driven precedence order.

The Taste-IV attributes mailbox and blackboard define the corresponding data-channel type. When the user specifies a data-channel, the ***DataChannelSize*** should be also defined forming the minimal size of the data type communicated via the blackboard or the mailbox (a non-blocking FIFO), in bytes. If the mailbox is provided, then the engineer should also define the **DataChannelLength** which determines the number of places the FIFO. An FPPN data-channel always exhibits two provided protected interfaces: channel-read and channel-write, while processes plug into the channels by required interfaces to read and write the channels. Note that if the user provides functional code to implement the channels then this code will be ignored by TASTE 2 BIP tools, and instead predefined BIP component types “mailbox” and “blackboard” are used. Therefore, in order to be consistent the user is currently supposed to paste the same code template (adapted for the given channel size, length and data type) for each “mailbox” and “blackboard”. This code is provided here via the tutorial example.

The example can be simulated in TASTE2BIP using the following steps:

```
> cd examples/sqexample
> taste-edit-interface-view&
# In Taste IV editor export TASTE-IV to XML (via “Tools” menu, overwrite existing xml file)
> taste2dolc.py
> cp -r fppn_REQUIRED_FILES/*  fppn
> cd fppn
> run
```

The taste2dolc.py program translates TASTE-IV XML file and functional code of processes (but not channels) to DOLC and puts the results into “fppn” folder. Then, by recursive (!) copying from folder “fppn_REQUIRED_FILES” we add some non-default files on top of the generated files. These files include “temp_asn1.h” where we imitate the ASN1 types. The three scripts that may be added by this copy step are:

(1) **run**
(2) **launch**
(3) **build-leon4**

The first script, **run**, executes TASTE2BIP with “rtbip” BIP RTE invoked in single-thread simulation mode. The time is simulated and the execution does not go in real-time mode (e.g. 1 s of simulated system time can take a fraction of a second to simulate in this mode).

The second script, **launch**, runs multi-threaded “multi-bip” BIP RTE compilation for *in real-time* on the linux workstation to imitate multi-core execution on the workstation.

The third script, **build-leon4**, builds for RTEMS-4.11 SMP for NGMP board. It links the application to “multi-bip” BIP RTE library compiled for very specific version of RTEMS. This build is likely to not work for any other version of RTEMS, it is recommended to build for exactly the same version via our virtual machine. Note that some examples may contain instead script **build-leon4down100**, which indicates that the given script runs the given example on NGMP with real time slow down factor 100 (for periods and deadlines). This is needed to run examples that print text to screen.

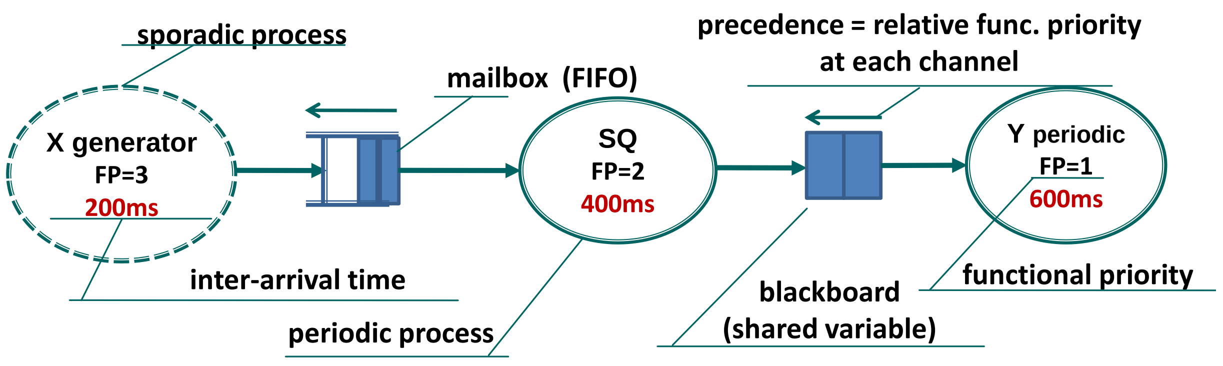
In the end we execute “run” script, but we could also have executed two others.

2.2 FPPN tutorial example: “sqexample”

In this section we introduce the “sqexample” (square-process example) provided with Taste2BIP. Note that there can be small discrepancies between the version described here and the version actually provided with the tools. We use this example to demonstrate:

1) TASTE modelling
2) TASTE 2 BIP and scheduler
3) BIP RTE deployment on RTEMS 4.11 on LEON4 platform.

The process network of the application is shown in Figure 2 represents an imaginary data processing application chain where “Xsporadic”, a sporadic process, generates values, the “Square”' process calculates the square of the previously generated values and the “Y” periodic function that serves as a sink for the squared value. A sporadic event (emulating command from the environment) activates the sporadic process “Xsporadic”, which is annotated by its minimal inter-arrival time is also shown. The periodic processes are annotated by their periods. The inter-process channels, the blackboard (shared variable) and a mailbox (FIFO) are also illustrated. The arc depicted above of each one of the inter-process channels indicates the relative functional priority. Additionally, the environment input/output channels are shown. In this example, the data flow in the channel goes in the opposite direction of the functional priority order. Figure 3 depicts the TASTE-IV model of the FPPN network described above.



X generator
Min Inter-arrival time = 200ms
Deadline=810
WCET = 10
Criticality = D

Mailbox
Size = 64 bits
Length = 8

SQ
Period = 400ms
Deadline = 390ms
WCET = 10
Criticality = E

blackboard
Size = 64 bits

Y periodic
Period = 600ms
Deadline = 590ms
WCET = 20
Criticality = E

Figure 2:Fixed Priority Process Network Example

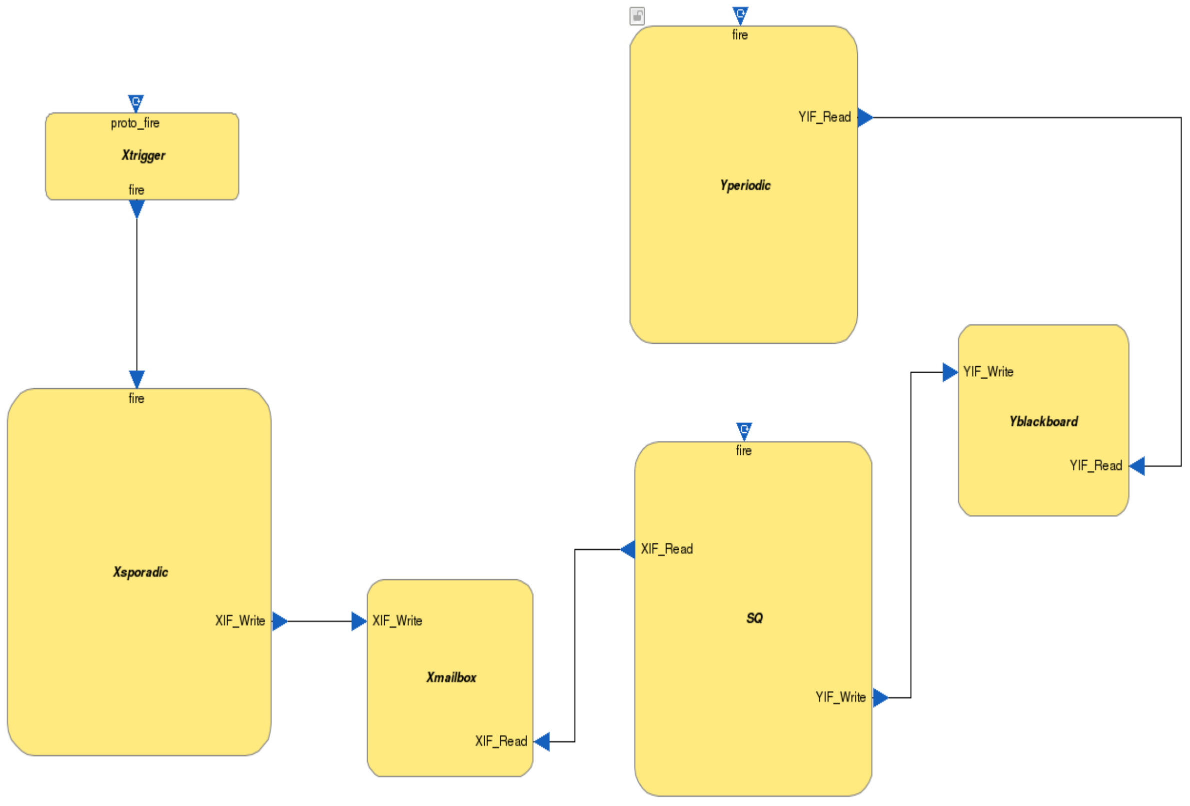


Figure 3: TASTE-IV Model of the FPPN Example

2.3 Process and Channel Modelling

Though the complete example is provided, in this section we give a tutorial how to reconstruct it from scratch.

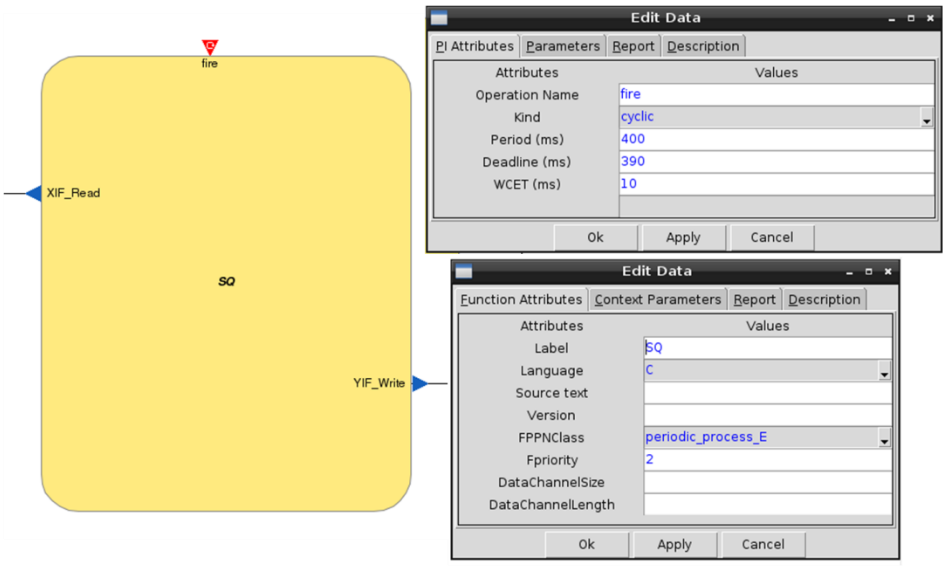
2.3.1. Creating a Periodic process

A periodic process is modelled (e.g. the SQ and Y periodic processes of Figure 2 example) in TASTE-IV with the combination of a TASTE-IV Function and a cyclic Provided Interface:

a) Create a TASTE-IV function (e.g. label = SQ, label=Yperiodic)
   I. Edit Properties.
   II. Set Language to be C (we currently support only C language for functional modelling)
   III. Select the corresponding FPPNClass of periodic type (i.e. periodic_process_A, periodic_process_B, periodic_process_C, periodic_process_D, periodic_process_E) according to the specification of criticality level.
   IV. Set the proper Fpriority integer value.

b) Create a Provided Interface with name "fire" of cyclic kind and set the attributes according to the specification (period, deadline and WCET values).

Figure 4 illustrates the SQ process node in TASTE-IV and its parameters.



Attributes	Values
Operation Name	fire
Kind	cyclic
Period (ms)	400
Deadline (ms)	390
WCET (ms)	10

Attributes	Values
Label	SQ
Language	C
Source text	
Version	
FPPNClass	periodic_process_E
Fpriority	2
DataChannelSize	
DataChannelLength	

Figure 4: SQ periodic model in TASTE-IV

2.3.2. Creating a Sporadic process

A sporadic process (e.g. the Xsporadic process of the example of Figure 2) is formed by connected two (2) functions in TASTE-IV and attaching two provided interfaces and one required-interface a shown in Figure 4. Specifically, the steps to be performed are:

a) Create a TASTE-IV function (e.g. label=Xsporadic) and configured it to perform as a sporadic process
   I. Edit Properties.
   II. Set Language to be C (we currently support C language for functional modelling)
   III. Select the corresponding FPPNClass of sporadic type (i.e. sporadic_process_A, sporadic_process_B, sporadic_process_C, sporadic_process_D, sporadic_process_E) according to the specification of criticality level
   IV. Set the proper Fpriority integer value.

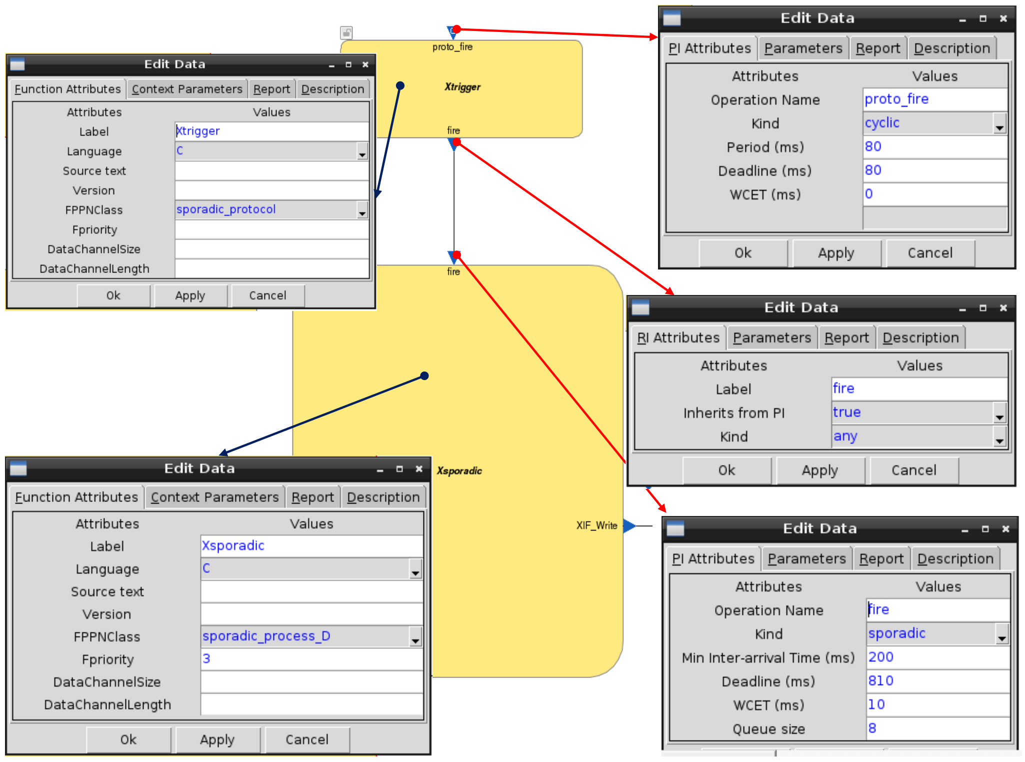
b) Create a TASTE-IV function (e.g. label=Xtrigger) and configured it to perform as a sporadic protocol that "fires" (i.e. calls the "fire" function) the associated sporadic process.
   I. Edit Properties.
   II. Set Language to be C (we currently support C language for functional modelling)

III. Select the corresponding FPPNClass of sporadic protocol (i.e. sporadic_protocol).

c) Connect a Provided Interface (e.g. operation name =proto_fire) of cyclic kind to the sporadic protocol and set the “fire” attributes according to the specification (period, deadline and WCET values).

d) Attach a Provided Interface (e.g. operation name = fire) of type "sporadic" to the sporadic process and set the PI attributes according to the specification (min-inter-arrival time, deadline, WCET, and Queue size values). The Queue size should correspond to the DataChannelLength of the connected data channel (Mailbox channel type).

e) Connect using a Required Interface the sporadic protocol (e.g. Xtrigger) with the Provided Interface exhibited by the sporadic process (e.g. Xsporadic).



Edit Data — Function Attributes | Context Parameters | Report | Description

Attributes	Values
Label	Xtrigger
Language	C
Source text	
Version	
FPPNClass	sporadic_protocol
Fpriority	
DataChannelSize	
DataChannelLength	

Edit Data — PI Attributes | Parameters | Report | Description

Attributes	Values
Operation Name	proto_fire
Kind	cyclic
Period (ms)	80
Deadline (ms)	80
WCET (ms)	0

Edit Data — RI Attributes | Parameters | Report | Description

Attributes	Values
Label	fire
Inherits from PI	true
Kind	any

Edit Data — Function Attributes | Context Parameters | Report | Description

Attributes	Values
Label	Xsporadic
Language	C
Source text	
Version	
FPPNClass	sporadic_process_D
Fpriority	3
DataChannelSize	
DataChannelLength	

Edit Data — PI Attributes | Parameters | Report | Description

Attributes	Values
Operation Name	fire
Kind	sporadic
Min Inter-arrival Time (ms)	200
Deadline (ms)	810
WCET (ms)	10
Queue size	8

Figure 5:Sporadic process configuration in TASTE-IV

2.3.3. Creating a Mailbox (FIFO) data channel

A mailbox data channel and its conveyed data parameters of its two Provided Interfaces are defined using TASTE IV(see Figure 6) as follows:

a) Create a TASTE-IV function (e.g. label = Xmailbox) and configured it to perform as a mailbox channel type.

I. Edit Properties.

II. Set Language to be C (we currently support C language for functional modelling)

III. Select the corresponding FPPNClass of mailbox type (i.e.mailbox).

IV. Set the DataChannelSize in bytes (practically to a large number e.g. 1024). It defines the size of the ASN1 data type that is conveyed using the channel.

V. Set the DataChannelLength to an integer value that defines the length of the FIFO.

b) Configure the write port by connecting a Provided Interface (e.g. operation name = XIF_Write) of protected kind to the mailbox channel and define the Deadline and WCET values to equal to the values of the process that is going to be connected with.

I. In this example the XIF_Write provided interface serves as input which writes the data to the channel and it is defined using three (3) parameters as follows:

i. The data parameter (e.g. write_data) is the data written to the channel and is defined by an ASN1 data type. In the depicted example, the write_data parameter is a double value of MyReal ASN1 data type. **Please note that this parameter should be the same for both write and read PIs**. Its type is MyReal, its Encoding Protocol is Native and its Direction is IN.

ii. The valid parameter corresponds to the validity of the write data. Its type is T_Boolean (ASN1 data type), its Encoding Protocol is Native and its Direction is IN.

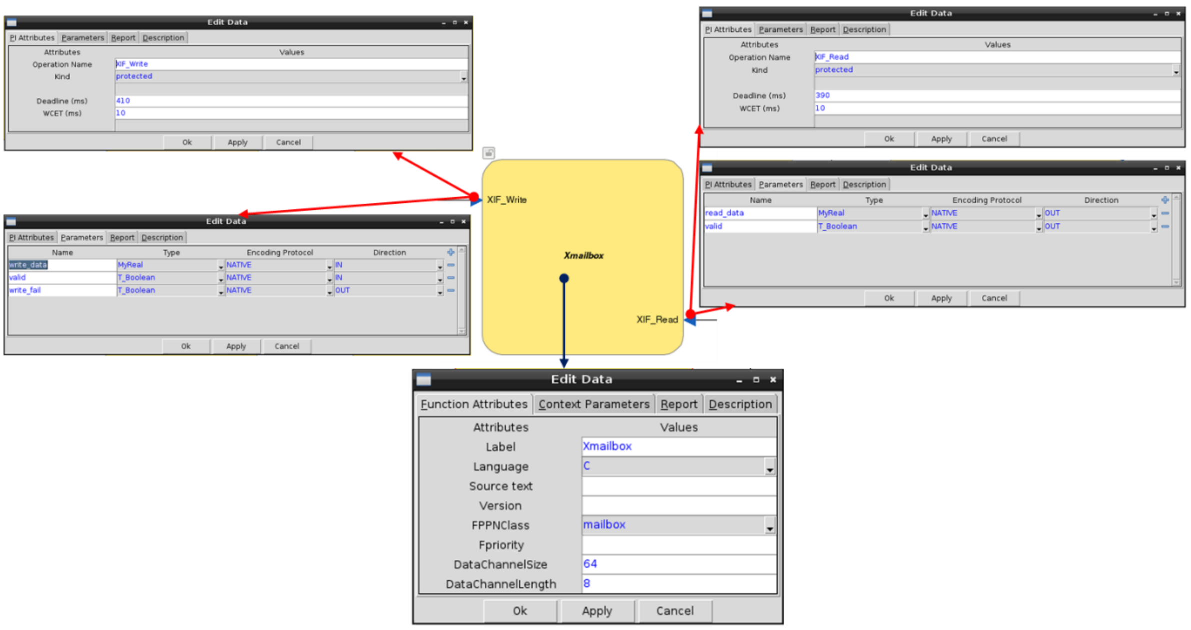
iii. The write_fail indicates if there is a fail during the write process to the channel. Its type is T_Boolean (ASN1 data type), its Encoding Protocol is Native and its Direction is OUT.

c) Configure the read port by connecting a Provided Interface (e.g. operation name = XIF_Read) of protected kind to the mailbox channel and define the Deadline and WCET values to equal to the values of the process that is going to be connected with.

I. In this example the XIF_Read provided interface serves as input which writes the data to the channel and it is defined using two (2) parameters as follows:

i. The data parameter (e.g. read_data) is the data read from the channel and is defined by an ASN1 data type. In the depicted example, the read_data parameter is a double value of MyReal ASN1 data type. **Please note that this parameter should be the same for both write and read PIs**. Its type is MyReal, its Encoding Protocol is Native and its Direction is OUT.

ii. The valid parameter corresponds to the validity of the read data. Its type is T_Boolean (ASN1 data type), its Encoding Protocol is Native and its Direction is OUT.



Attributes	Values
Operation Name	XIF_Write
Kind	protected
Deadline (ms)	410
WCET (ms)	10

Name	Type	Encoding Protocol	Direction
write_data	MyReal	NATIVE	IN
valid	T_Boolean	NATIVE	IN
write_fail	T_Boolean	NATIVE	OUT

Attributes	Values
Operation Name	XIF_Read
Kind	protected
Deadline (ms)	390
WCET (ms)	10

Name	Type	Encoding Protocol	Direction
read_data	MyReal	NATIVE	OUT
valid	T_Boolean	NATIVE	OUT

Attributes	Values
Label	Xmailbox
Language	C
Source text	
Version	
FPPNClass	mailbox
Fpriority	
DataChannelSize	64
DataChannelLength	8

Figure 6: Mailbox data channel definition in TASTE-IV

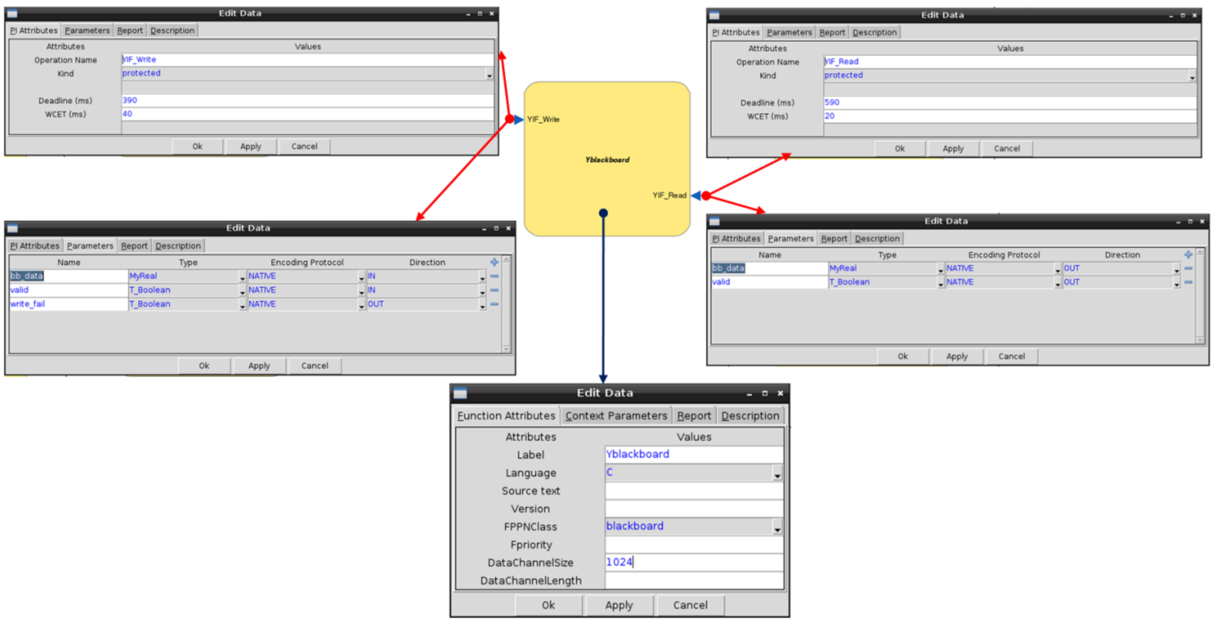
2.3.4. Blackboard (shared variable) data channel

A blackboard data channel and its conveyed data parameters of its two Provided Interfaces are defined using TASTE IV(see Figure 7) as follows:

a) Create a TASTE-IV function (e.g. label = Yblackboard) and configured it to perform as a blackboard channel type.
   I. Edit Properties.
   II. Set Language to be C (we currently support C language for functional modelling)
   III. Select the corresponding FPPNClass of blackboard type (i.e.blackboard).
   IV. Set the DataChannelSize in bytes (practically to a large number e.g. 1024). It defines the size of the ASN1 data type that is conveyed using the channel. The lifespan of the blackboard equal to 32768 (it is defined only for the case of a blackboard data channel and it is generated automatically by the TASTE2DOLC generator). Lifespan in DOLC is not a time-counter but a down-counter. It is considered when the writer writes once and after that it does not write anymore. It counts the rounds of the writer without writing and when it reaches lifespan value the contents of blackboard is erased (invalidated).

b) Configure the write port by connecting a Provided Interface (e.g. operation name = YIF_Write) of protected kind to the mailbox channel and define the Deadline and WCET values to equal to the values of the process that is going to be connected with.
   I. In this example the XIF_Write provided interface serves as input which writes the data to the channel and it is defined using three (3) parameters as follows:
      i. The data parameter (e.g. bb_data) is the data written to the channel and is defined by an ASN1 data type. In the depicted example, the bb_data parameter is a double value of MyReal ASN1 data type. **Please note that this parameter should be the same for both write and read PIs**. Its type is MyReal, its Encoding Protocol is Native and its Direction is IN.
      ii. The valid parameter corresponds to the validity of the write data. Its type is T_Boolean (ASN1 data type), its Encoding Protocol is Native and its Direction is IN.
      iii. The write_fail indicates if there is a fail during the write process to the channel. Its type is T_Boolean (ASN1 data type), its Encoding Protocol is Native and its Direction is OUT.

c) Configure the read port by connecting a Provided Interface (e.g. operation name = YIF_Read) of protected kind to the mailbox channel and define the Deadline and WCET values to equal to the values of the process that is going to be connected with.
   I. In this example the XIF_Read provided interface serves as input which writes the data to the channel and it is defined using two (2) parameters as follows:
      i. The data parameter (e.g. bb_data) is the data read from the channel and is defined by an ASN1 data type. In the depicted example, the bb_data parameter is a double value of MyReal ASN1 data type. **Please note that this parameter should be the same for both write and read PIs**. Its type is MyReal, its Encoding Protocol is Native and its Direction is OUT.
      ii. The valid parameter corresponds to the validity of the read data. Its type is T_Boolean (ASN1 data type), its Encoding Protocol is Native and its Direction is OUT.



Edit Data — PI Attributes | Parameters | Report | Description

Attributes	Values
Operation Name	YIF_Write
Kind	protected
Deadline (ms)	390
WCET (ms)	40

Edit Data — PI Attributes | Parameters | Report | Description

Name	Type	Encoding Protocol	Direction
bb_data	MyReal	NATIVE	IN
valid	T_Boolean	NATIVE	IN
write_fail	T_Boolean	NATIVE	OUT

Edit Data — PI Attributes | Parameters | Report | Description

Attributes	Values
Operation Name	YIF_Read
Kind	protected
Deadline (ms)	590
WCET (ms)	20

Edit Data — PI Attributes | Parameters | Report | Description

Name	Type	Encoding Protocol	Direction
bb_data	MyReal	NATIVE	OUT
valid	T_Boolean	NATIVE	OUT

Edit Data — Function Attributes | Context Parameters | Report | Description

Attributes	Values
Label	Yblackboard
Language	C
Source text	
Version	
FPPNClass	blackboard
Fpriority	
DataChannelSize	1024
DataChannelLength	

Figure 7:Blackboard data channel definition in TASTE-IV

2.3.5. Finalizing the Design

After having created the process network structure in TASTE, the user defines the ASN1 data types in DATA-VIEW; see, however, the later remark on limitations of using ASN with TASTE2BIP. Then, if the user desires to pass the design also through the build script of TASTE tools then the user has to configure the TASTE-DV (Deployment View) model by binding the created functions to the specified processor (typically to linux x86 partition). Then the user can, via TASTE-IV GUI, generate code skeletons (*.c files for each functional block). From these skeletons functional code is created, this procedure is explained in the next subsection using our tutorial example. After introducing the functional code, the user can execute TASTE 2 BIP/scheduling toolchain.

Before proceeding to TASTE 2 BIP translation, the user has to either finish and validate the functional code in TASTE or to create at least partial functional code, in particular one has to introduce at least the periodic/sporadic process code responsible for the accesses to the data channels, i.e. read and write function calls and the variables that serve as input and output arguments to them. Also for each process the user should create "local state" data structure responsible to hold principal variables that contain local process state preserved from one process firing to the next one. After adding this code and ensuring that it compiles, the user can start generating his first BIP models by TASTE2BIP conversion. For this he/she has to (1) export the IV design to XML (i.e. InterfaceView.xml file generation via editor GUI) and (2) execute taste2dolc.py (3) execute one of the three scripts as provided in the example (run, launch or build for embedded platform).

When translating the C code from TASTE-IV to DOLC the tool makes slight changes in the C coding, but most part is preserved intact. One important change is that the TASTE-generated header files are not preserved but replaced by special DOLC-style header files and the link to the header files that contain the ASN-compiler generated data types is not preserved and replaced by a link to "temp_asn1.h", which is supposed to serve to manually include or specify ASN data types. The user has to reestablish the link by adjusting the C source code and Makefiles. In our examples, due to lack of experience with ASN 1 data type language and in order to reuse the legacy types already provided in the use case code we manually specify the communicated data types in "temp_asn1.h" instead of programming them in Data View and translating them to C.

Note that external C/C++ libraries/headers are linked to the functional code in TASTE and in DOLC2BIP using two different methods, and TASTE 2 DOLC converter currently does not translate one way of linking to another.

The C based functional model of the FPPN application in TASTE-IV can be developed on the generated code skeletons based on the C FPPN guidelines/examples given in this section. After a successful build in TASTE-IV the engineer invokes the generated executable and runs the functional simulation of the application using TASTE tools. Within this context, a first evaluation of the functional behavior of the system under development is simulated without incorporation of the functional priority order.

2.3.6. Writing the Functional Code for FPPN Model

2.3.6.1. FPPNClass "sporadic-protocol" Coding

i. Sporadic protocol (such as Xtrigger in our example) is executed periodically at relatively high frequency (80ms period in our example) and polls the environment for application-specific condition to activate the process. When the condition is satisfied it issues a call the sporadic process via asynchronous function call. Our current version does not support passing input arguments to the sporadic process. In TASTE2BIP framework the sporadic protocol is considered to be not one of the tasks but instead part of runtime overhead (the "delta" parameter in scheduler, explained later) and is accounted as such in the scheduler.

   In "sqexample" we implemented an arbitrary protocol that ensures aperiodic activation of process Xsporadic. To reproduce it, we first define the local state variables of the protocol in the generated *.c file. In our tutorial example the file is xtrigger.c, located at the xtrigger folder. We define the local state variables as members of a struct with special naming conventions:

```
struct 'lower-case(TASTE-IV function label)'_state_t {
  int trigger_counter;
  int proto_counter;
  int call_counter;
  int step;
};
static struct 'lower-case(TASTE-IV function label)'_state_t state;
```

   Note: use lower case letters in the structure definition. The **name** of the TASTE-IV function (i.e. the label in TASTE-IV function attributes) **should be included in** the structure definition. In our example the label of the TASTE-IV function of the sporadic-protocol is Xtrigger, and therefore the structure definition becomes:

```
struct xtrigger_state_t {
  int trigger_counter;
  int proto_counter;
  int call_counter;
  int step;
};
static struct xtrigger_state_t state;
```

ii. For ensuring proper initialization of the local state, placing the following C code in the body of the startup() function.
In our case, the result is:

```
void xtrigger_startup()
{
   /* Write your initialization code here,
     but do not make any call to a required interface. */
   state.call_counter = 0;
   state.proto_counter = 0;
   state.step = 1;
   state.trigger_counter = 0;
}
```

iii. Configure the body of the C fire() function accordingly. In our case it becomes:

```
void xtrigger_PI_proto_fire()
{
   /* Write your code here! */
   int time  = 80*state.proto_counter;

   if ((state.proto_counter/5) == state.call_counter) {
    if ((state.proto_counter%5) == state.trigger_counter) {
       if (state.trigger_counter == 4) {
         state.step = -1;
       }

       if (state.trigger_counter == 0) {
         state.step = +1;
       }

       state.trigger_counter += state.step;

       // trigger!
       printf("---------------------------------------\n");
       printf("Time: [%d] (tr)\n", time);
       xtrigger_RI_fire();//It connects with  the sporadic process
state.call_counter++;
     }
   }
   state.proto_counter++;
}
```

2.3.6.2. FPPNClass "mailbox" Coding

Recall that TASTE 2 BIP tool suite currently does not convert the C code for mailbox and blackboard channels into BIP but instead substitutes standard BIP components. The C code for mailbox given here emulates the behavior of its BIP component.

i. To reproduce mailbox of "sqexample" first define the local state structure and channel length in the mailbox's *.c file (in our tutorial example the file is xmailbox.c and is located at the xmailbox folder) as follows:

```
#define MB_LENGTH 8// length of the FIFO
```

```
typedef asn1SccMyReal MB_TYPE;

struct xmailbox_contents_t {
  MB_TYPE buff_data[MB_LENGTH];
  int rpos, wpos, tokens;
};

static struct xmailbox_contents_t  mb_contents;
```

The mailbox is implemented here as a classical circular buffer for FIFO behavior.

ii. Ensure the proper local state initialization by placing the following C code in the body of the startup() function:

```
void xmailbox_startup()
{
   /* Write your initialization code here,
     but do not make any call to a required interface. */
   mb_contents.rpos   = 0;
   mb_contents.wpos   = 0;
   mb_contents.tokens = 0;
}
```

iii. Configure the body of the mailbox channel writing subroutine accordingly. In our case it becomes:

```
void    xmailbox_PI_XIF_Write(const    asn1SccMyReal    *IN_write_data,    const
asn1SccT_Boolean *IN_valid, asn1SccT_Boolean *OUT_write_fail)
{
   /* Write your code here! */
   /*assumption of TASTE*/
   int len = 1;
   /* Write your code here! */
   struct xmailbox_contents_t* p = &mb_contents;
   if (*IN_valid) {
      int elements_to_write = len;
*OUT_write_fail = false;
      if (MB_LENGTH elements_to_write > MB_LENGTH) {
*OUT_write_fail =  true;
elements_to_write = MB_LENGTH – p->tokens;
}

int k;
for ( k=0; k < elements_to_write; k++, p->wpos=(p->wpos+1)%MB_LENGTH) {
p->buff_data[p->wpos] = IN_write_data[k];
}
p->tokens = p->tokens + elements_to_write;

   } else {
    // writing invalid data=>erasing
    int elements_to_erase = len;
    *OUT_write_fail = 0;
    if (elements_to_erase > p->tokens) {
*OUT_write_fail =  true;//too many: elements_to_erase - p->tokens;
elements_to_erase = p->tokens;
    }
    p->tokens = p->tokens - elements_to_erase;
    p->wpos = ((p->wpos + MB_LENGTH) - elements_to_erase) % MB_LENGTH;
```

```
  }
}
```

iv. Configure the body of the C Read () function accordingly. In our case it becomes:

```
void xmailbox_PI_XIF_Read(asn1SccMyReal *OUT_read_data, asn1SccT_Boolean
*OUT_valid)
{
   /* Write your code here! */
   int len = 1;

   /* Write your code here! */
   struct xmailbox_contents_t* p = &mb_contents;
   if (p->tokens < len) {
     *OUT_valid = false;
   } else {
     *OUT_valid = true;
     int k;
     for ( k=0; k < len; k++, p->rpos=(p->rpos+1)%MB_LENGTH) {
         OUT_read_data[k] = p->buff_data[p->rpos];
     }
     p->tokens = p->tokens - len;
   }
}
```

2.3.6.3. *FPPNClass “blackboard” Coding*

Recall that TASTE 2 BIP tool suite currently does not convert the C code for mailbox and blackboard channels into BIP but instead substitutes standard BIP components. The C code for blackboard given here emulates the behavior of its BIP component.

i. To reproduce blackboard of “sqexample” first define the local state structure in the generated *.c file (in our tutorial example the file is yblackboard.c and is located at the yblackboard folder) as follows:

```
struct yblackboard_contents_t {//yblackboard is the label of TASTE-IV function
attribute

  bool validity;
  asn1SccMyReal bbdata;
};
static struct yblackboard_contents_t bb_contents;
```

ii. Initialise the local state by placing the following C code in the body of the startup() function:

```
void yblackboard_startup()
{
/* Write your initialization code here,
     but do not make any call to a required interface. */
   bb_contents.validity = false;
}
```

iii. Configure the body of the channel writing function accordingly. In our case it becomes:

```
void yblackboard_PI_YIF_Write(const asn1SccMyReal *IN_bb_data, const
asn1SccT_Boolean *IN_valid, asn1SccT_Boolean *OUT_write_fail)
```

```
{
/* Write your code here! */

   bb_contents.bbdata = *IN_bb_data;
   bb_contents.validity = *IN_valid;
   *OUT_write_fail = false; // impossible to fail writing to blackboard
}
```

iv. Configure the body of the channel reading function accordingly. In our case it becomes:

```
void yblackboard_PI_YIF_Read(asn1SccMyReal *OUT_bb_data, asn1SccT_Boolean
*OUT_valid)
{
/* Write your code here! */
   *OUT_bb_data  = bb_contents.bbdata;
   *OUT_valid  = bb_contents.validity;
}
```

2.3.6.4. FPPNClass "sporadic/periodic process" Coding

i. Define the local state structure in the generated *.c file (in our tutorial example the file is sq.c and is located at the sq folder) as follows:

```
struct sq_state_t {//sq is the label of TASTE-IV function given to this process
converted to lowercase
  int index;
  int length;
};

static struct sq_state_t state;
```

ii. Initialise accordingly the local state by placing the following C code in the body of the startup() function:

```
void sq_startup()
{
   /* Write your initialization code here,
     but do not make any call to a required interface. */
   state.index = 0;
   state.length = 200;
}
```

iii. Configure the body of the C fire() function accordingly. In our case it becomes:

```

void sq_PI_fire()
{
   /* Write your code here! */
   asn1SccMyReal x, y;
   asn1SccT_Boolean x_valid,y_valid, ywrite_fail;

  if (state.index < state.length)
  {
sq_RI_XIF_Read(&x, &x_valid);
if(x_valid == true)
{
  printf("--------------------------------------\n");
  printf("SQ function reads %f from X generator\n", (float)x);
y = x * x;
y_valid = true;
```

```
                        sq_RI_YIF_Write(&y, &y_valid, &ywrite_fail);


                         if (ywrite_fail != true)
                        {
                            printf("--------------------------------------\n");
                              printf("SQ function wrote %f\n", (float) y);
                         }
                          else
                         {
                              printf("--------------------------------------\n");
                        printf("SQ function FAILED to write %f!\n", (float) y);
                         }

                        }
                        else
                        {
                        printf("--------------------------------------\n");
                        printf("SQ function FAILED to read data from X generator\n");
                        }

}
 state.index++;
}
```

iv. Sporadic process ("xsporadic") is coded in a similar way as periodic one. In future work we will consider adding to the fire function the input arguments passed from the sporadic protocol that (asynchronously) calls the given sporadic process.

v. Here we have given the code for the "sq" process example, the code for other processes can be found in the example provided with the tools.

2.3.7. Important remarks

A. Our static scheduler and static task-graph generation option of TASTE 2 DOLC require that each sporadic process should connect to a single periodic process, called its "user process", through a single channel. In our running example, process "sq" is the user of "Xsporadic". In addition, the sporadic process should not connect to any other process. We put this restriction to avoid different sporadic processes to communicate, in which case precedence constraints between them would be dynamic, which would make static offline scheduling and static task-graph generation problematic. The deadline of sporadic processes should be larger than the periods of its user process (preferably, double the period of the user process).

B. The "DataChannelSize" attribute of a data-channel functional block is the size of the data type communicated via the channel, in bytes. This size is platform-dependent, but the user can also specify an upper bound that holds in any platform. In fact, this attribute must be redundant, as the ASN1 data type itself is specified in the arguments to the calls to data channel interfaces. Currently TASTE2BIP needs this attribute but in future work it will be possibly deprecated.

C. The functional code of the data channels is currently not included into TASTE 2 BIP translation, this is future work. The BIP data-channel component is taken from the component library based on the channel type.

D. A data-channel always includes two provided interfaces while processes only exhibit required interfaces to the channels for writing or reading the data over the channels.

E. TASTE2DOLC generator takes into account the FPPNClass attribute of every functional block to determine how to translate it. Among the additional attributes, the tool uses only those that are necessary for the model transformation procedure. For example if FPPNClass = “blackboard”, TASTE2DOLC generator will ignore the attributes “Fpriority” and “DataChannelLength”.

F. Each process has always a unique functional priority integer number. When two processes are compared by their functionally priority, we define that the process with the larger assigned integer numeric value of priority has the lower priority. At the translation to DOLC the functional priorities are taken into account to insert “precedence chain” arcs between pairs of processes that communicate via it least one data channel. The precedence goes from the process with higher priority to the one with the lower priority. The user can manually insert into DOLC XML file new precedence arcs between non-communicating processes to add scheduling-order constraints. These arcs should not, however, introduce a cyclic path. Note that unlike priority constraints in concurrency view, precedence constraints define scheduling order in a multi-core platform even between two processes mapped to different cores.

CHAPTER 3. TASTE to DOLC toolset

3.1 Installation of TASTE2DOLC generator

The TASTE2DOLC generator is developed in python 2.7 and consists of three files i.e.: a) **taste2dolc.py**, b) **fppnxml.py**, and c) **genmodules.py**. See in the tools distribution:

```
exe/taste2dolc
```

File "sourceme.rc" generated by the installation script (see earlier section Installation of tools) defines the extension of PATH variable to the path to taste2dolc.py so that the tool can be invoked just by typing "taste2dolc.py" inside the main folder of the TASTE design.

After the successful compilation and functional simulation of the FPPN model in TASTE-IV, export the interface view in xml format and generate the file InterfaceView.xml.

Note: Do not rename the InterfaceView.xml file or export it under a different name or directory due to the fact that TASTE2DOLC generator assumes this default name of the file at the default export directory.

3.2 Using TASTE2DOLC generator

The TASTE-IV to DOLC model transformation is performed using the TASTE2DOLC generator as follows by running **taste2dolc.py** in the main folder of TASTE design. Figure 8 depicts the inputs of the TASTE2DOLC generator and the produced outputs.

Note that the tool takes optional offsets.dat, a text file that specifies offset for each process:

```
<process1>=<offset_in_ms1>
<process2>=<offset_in_ms2>
….
```

These offsets would be taken into account in the task graph and the scheduler tool. However, to take these offsets to the online deployment requires manual adjustments of BIP models which we do not describe here. An example is given in GNC application manual in Use Case Technical Note [RefUseCase].

If **taste2dolc** terminates successfully a folder named *fppn* is created. The *fppn* folder contains the *src* folder that includes c-to-c transformed files (*.h and *.c files). **Note** that a c header file *temp_asn1.h* is also generated. This file defines the ASN1 data types used in TASTE-IV Data-View and is shown below. Currently, it automatically generates the following ASN1 type definitions while the user is requested to update it accordingly in case that more ASN1 data types are used in the TASTE-IV design.

Generated *temp_asn1.h* file.

```
#ifndef TEMP_ASN_H

#define TEMP_ASN_H

typedef int asn1SccMyInteger;

typedef int asn1SccMyReal;

typedef int asn1SccT_Boolean;
```

```
        #endif
```

Furthermore, the fppn folder also contains the *fppn-dolc.xml*, *generator.out*, *fppn_tg.jobs*, and the *fppn_tg.jobsprocs* files. The whole generation procedure is performed transparently to the user.

- The *fppn-dolc.xml* file defines in the xml format the DOL Critical description of the FPPN network and forms the output of the xml-to-xml model transformation performed by the TASTE2DOLC generator when the TASTE-IV xml functional model is fed to the latter. The *fppn-dolc.xml* file includes the process definitions, their associated controllers, the data channel definitions, the precedence chains per pair of processes that connected uniquely by a single channel, and the connections between a) channels and processes, and b) processes and channels.
- The *generator.out* contains logging information about the TASTE2DOLC generator execution steps while the other two define the task graph of the FPPN model.
- Specifically, *fppn_tg.jobs* file (task-graph) defines the jobs per process that take place within a Hyper-period *H* which is the LCM (these are the jobs in a unrolled schedule within the time-frame of least-common multiple of the periods). One job corresponds to one call to the "fire" function. The task-graph file (*.jobs) consists of 5 to 8 text columns depending on the maximal criticality level of the tasks.

  T***ask-graph file*** generated for *parallelprocs* example:

  ```
  0    25   2    12   12
  0    25   2    1    1
  0    25   1    6    6
  (J2, J3)
  (J2, J1)
  ```

  The task graph arcs (pair of jobs) are located below the definition of the jobs' characteristics (each line of the file corresponds to a job *id* (i) where *id* is the line number).
  The arcs are preceded by job description, one line per job with several columns of text.
  The first column of the file defines for each job *i* the inter-arrival time A_i (task period * time index of job execution i.e. i =0, 1, 2 ,3, 4…). The second column describes for each job *i* the absolute deadline D_i, where $D_i = min(H, A_i + DP_i)$. $DP_i$is the deadline of the process (task) of job *i*, A_i is the inter-arrival time defined in column A_i. Note that $D_i<=H$. It should be also noted that if a time-offset is defined for a specific process in the offsets.dat file then the TASTE2DOLC generator sets the A_i that corresponds to the first job definition of this specific process to be equal to the time-offset rather than zero ms. The third column defines the criticality levels of each job. It is defined that A->5, B->4, C->3, D->2, E->1. The criticality levels are assigned according to 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. Mathematically it is measured by tolerated error rate per hour of flight, 10^{-9} for autopilot and say 10^{-3} for the entertainment of board. The forth to eighth columns repeat the WCET value of the process for which the job *i* is defined. The number of WCET repetitions (columns) is specified according to the formula *WCET column repetitions = max(2, max(criticality levels of all processes))*. This dictates that the minimum number of WCET column is equal to two and the maximum is equal to five. The rest of the file contains the pair of jobs arcs.
- The file *fppn_tg.jobsprocs* defines the number of jobs executed by each process (task).

Important note: In case of the presence of a sporadic process in the FPPN model the task graph generation involves the following steps, which are performed *automatically* by the TASTE2DOLC generator.

- The minimal inter-arrival time of the sporadic process defined in TASTE-IV should be larger or equal to the period of its user process. To this end the generator assigns in DOLC xml file the minimal inter-arrival of the sporadic process (interval) = Period of the user process.
- If *m= ceiling(Period of user/ minimal inter-arrival time of sporadic)>1*(*m* is the burst size), theTASTE2DOLCgenerator generate *m* jobs per each inter-arrival time defined in the first column for the sporadic process.
- When we generate task graph we temporarily replace the sporadic process by imaginary periodic process that:
    - Has larger functional priority than the user process.
    - Has the same period as the user process.
    - Has deadline that is equal to the deadline of the sporadic process minus the period of the user process.

- After the substitution of the sporadic processes by imaginary processes, an imaginary process network is obtained where all processes are periodic processes, and the same task-graph generation procedure is performed as for the periodic ones. Note that this replacement of sporadic by imaginary is needed only when we generate the task graph.

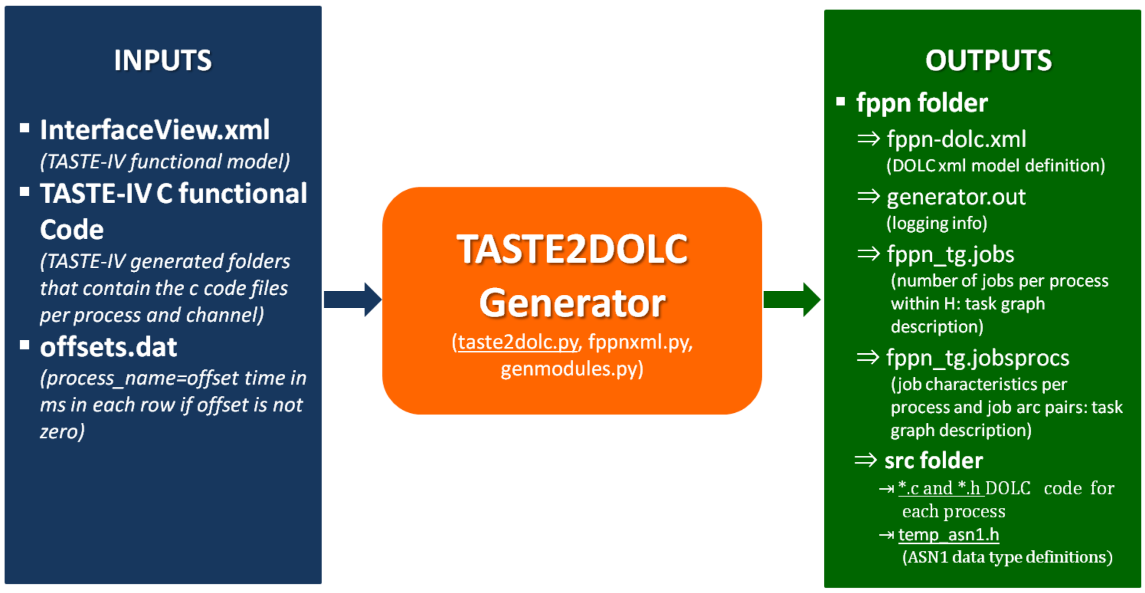

INPUTS
InterfaceView.xml
(TASTE-IV functional model)
TASTE-IV C functional Code
(TASTE-IV generated folders that contain the c code files per process and channel)
offsets.dat
(process_name=offset time in ms in each row if offset is not zero)
TASTE2DOLC Generator
(taste2dolc.py, fppnxml.py, genmodules.py)
OUTPUTS
fppn folder
⇒ fppn-dolc.xml
(DOLC xml model definition)
⇒ generator.out
(logging info)
⇒ fppn_tg.jobs
(number of jobs per process within H: task graph description)
⇒ fppn_tg.jobsprocs
(job characteristics per process and job arc pairs: task graph description)
⇒ src folder
*.c and *.h DOLC code for each process
temp_asn1.h
(ASN1 data type definitions)

Figure 8: TASTE2DOLC generator

3.3 Results for the "sqexample" benchmark

Go to the sqexample folder and execute:

```
> taste-edit-interface-view&
# In Taste IV editor export TASTE-IV to XML
> taste2dolc.py
> cp -r fppn_REQUIRED_FILES/*  fppn
> cd fppn
```

Extract the sqexample.tar.gz file to a specific folder (e.g. /home/assert/sqexample) on your workstation. This folder includes the FPPN example of Figures 2 and 3. It also includes the TASTE2DOLC generator. Guide through the files included in the extracted folder and open the TASTE-IV model and execute the function simulation. Execute in a terminal the command ./taste2dolc.py under the sqexample directory. The TASTE2DOLC generator creates under the fppn folder the following items (Figure 9 depicts the task-graph of the example).

The following generated files are present there:

- **fppn-dolc.xml**

```
<?xml version="1.0"?>
<app name="TASTE2DOLC_XML">

<!--Sporadic Process: xsporadic-->
<process name="xsporadic" criticality="D">
        <source location="xsporadic.c"/>
        <port name="p_PROC_xsporadic_start" type="in_event"><event name="start"/></port>
        <port name="p_PROC_xsporadic_finish" type="out_event"><event name="finish"/></port>
        <!--data port connected to channel xmailbox-->
        <port name="p_PROC_XIF_Write" type="out_data"/>
</process>

<!--Aperiodic Controller of aperiodic process: xsporadic-->
<controller name="Cxsporadic" deadline="0.81">
        <activation type="aperiodic">
                <parameter name="m_max" value="2"/>
                <parameter name="interval" value="0.4"/>
                <parameter name="protocol_period" value="0.08"/>
                <parameter name="protocol_source" value="xtrigger.c"/>
        </activation>
        <port name="p_CTRL_xsporadic_start" type="out_event"><event name="start"/></port>
        <port name="p_CTRL_xsporadic_finish" type="in_event"><event name="finish"/></port>
</controller>

<!--Control-channel from aperiodic process to aperiodic controller-->
<control_channel name="Ctrl_xsporadic_p2c">
        <port name="p_PROC_xsporadic_finish"/>
        <port name="p_CTRL_Xsporadic_finish"/>
</control_channel>

<!--Control-channel from aperiodic controller to aperiodic process -->
<control_channel name="Ctrl_xsporadic_c2p">
        <port name="p_CTRL_xsporadic_start"/>
        <port name="p_PROC_xsporadic_start"/>
</control_channel>

<!--Mailbox data-channel: xmailbox-->
<data_channel name="xmailbox" type="mailbox" size="64" length="8">
        <port name="p_CHAN_XIF_Write" type="in_data"/>
        <port name="p_CHAN_XIF_Read" type="out_data"/>
</data_channel>

<!--Periodic Process: SQ-->
<process name="sq" criticality="E">
        <source location="sq.c"/>
        <port name="p_PROC_sq_start" type="in_event"><event name="start"/></port>
        <port name="p_PROC_sq_finish" type="out_event"><event name="finish"/></port>
        <!--data port connected to channel xmailbox-->
        <port name="p_PROC_XIF_Read" type="in_data"/>
        <!--data port connected to channel yblackboard-->
        <port name="p_PROC_YIF_Write" type="out_data"/>
</process>

<!--Periodic Controller of process: sq-->
<controller name="Csq" deadline="0.39">
        <activation type="periodic">
        <parameter name="period" value="0.4"/>
        </activation>
        <port name="p_CTRL_sq_start" type="out_event"><event name="start"/></port>
        <port name="p_CTRL_sq_finish" type="in_event"><event name="finish"/></port>
</controller>

<!--Control-channel from process sq to controller-->
<control_channel name="Ctrl_sq_p2c">
        <port name="p_PROC_sq_finish"/>
        <port name="p_CTRL_sq_finish"/>
</control_channel>

<!--Control-channel from controller to process SQ-->
<control_channel name="Ctrl_sq_c2p">
        <port name="p_CTRL_sq_start"/>
        <port name="p_PROC_sq_start"/>
</control_channel>

<!--Blackboard data-channel: yblackboard-->
<data_channel name="yblackboard" type="blackboard" size="1024" lifespan="32768">
        <port name="p_CHAN_YIF_Write" type="in_data"/>
        <port name="p_CHAN_YIF_Read" type="out_data"/>
</data_channel>

<!--Periodic Process: Yperiodic-->
<process name="yperiodic" criticality="E">
        <source location="yperiodic.c"/>
        <port name="p_PROC_yperiodic_start" type="in_event"><event name="start"/></port>
        <port name="p_PROC_yperiodic_finish" type="out_event"><event name="finish"/></port>
        <!--data port connected to channel yblackboard-->
        <port name="p_PROC_YIF_Read" type="in_data"/>
</process>

<!--Periodic Controller of process: yperiodic-->
<controller name="Cyperiodic" deadline="0.59">
        <activation type="periodic"><parameter name="period" value="0.6"/></activation>
        <port name="p_CTRL_yperiodic_start" type="out_event"><event name="start"/></port>
        <port name="p_CTRL_yperiodic_finish" type="in_event"><event name="finish"/></port>
</controller>

<!--Control-channel from process yperiodic to controller-->
<control_channel name="Ctrl_yperiodic_p2c">
        <port name="p_PROC_yperiodic_finish"/>
        <port name="p_CTRL_yperiodic_finish"/>
</control_channel>

<!--Control-channel from controller to process Yperiodic-->
<control_channel name="Ctrl_yperiodic_c2p">
        <port name="p_CTRL_yperiodic_start"/>
        <port name="p_PROC_yperiodic_start"/>
</control_channel>

<!--Data connection -->
<connection name="xmailbox_channel_IN">
        <port name="p_PROC_XIF_Write"/>
        <port name="p_CHAN_XIF_Write"/>
</connection>

<!--Data connection -->
<connection name="xmailbox_channel_OUT">
        <port name="p_PROC_XIF_Read"/>
        <port name="p_CHAN_XIF_Read"/>
</connection>

<!--Data connection -->
<connection name="yblackboard_channel_IN">
        <port name="p_PROC_YIF_Write"/>
        <port name="p_CHAN_YIF_Write"/>
</connection>

<!--Data connection-->
<connection name="yblackboard_channel_OUT">
```

```
            <port name="p_PROC_YIF_Read"/>
            <port name="p_CHAN_YIF_Read"/>
</connection>

<!--Global requirements-->
<global name="globalCycle">
            <precedence chain="sq,xsporadic"/>
            <precedence chain="yperiodic,sq"/>
            </global>
</app>
```

- **fppn_tg.jobs**

Extra column in the left show for convenience the file line numbers, which correspond to job numbers J1-J11.

```
1  | 0         410       2         10        10
2  | 0         410       2         10        10
3  | 400       810       2         10        10
4  | 400       810       2         10        10
5  | 800       1200      2         10        10
6  | 800       1200      2         10        10
7  | 0         390       1         10        10
8  | 400       790       1         10        10
9  | 800       1190      1         10        10
10 | 0         590       1         20        20
11 | 600       1190      1         20        20
12 | (J1, J2)
13 | (J2, J7)
14 | (J7, J3)
15 | (J3, J4)
16 | (J4, J8)
17 | (J8, J5)
   | (J5, J6)
   | (J6, J9)
   | (J10, J7)
   | (J7, J8)
   | (J8, J11)
   | (J11, J9)
   | (J2, J3)
   | (J4, J5)
   | (J8, J9)
   | (J10, J11)
```

- **fppn_tg.jobsprocs**

```
xsporadic      6
sq             3
yperiodic      2
```

This file implies that:

- J1-J6 belong to “xsporadic”
- J7-J9 belong to “sq”
- J10, J11 belong to “yperiodic”

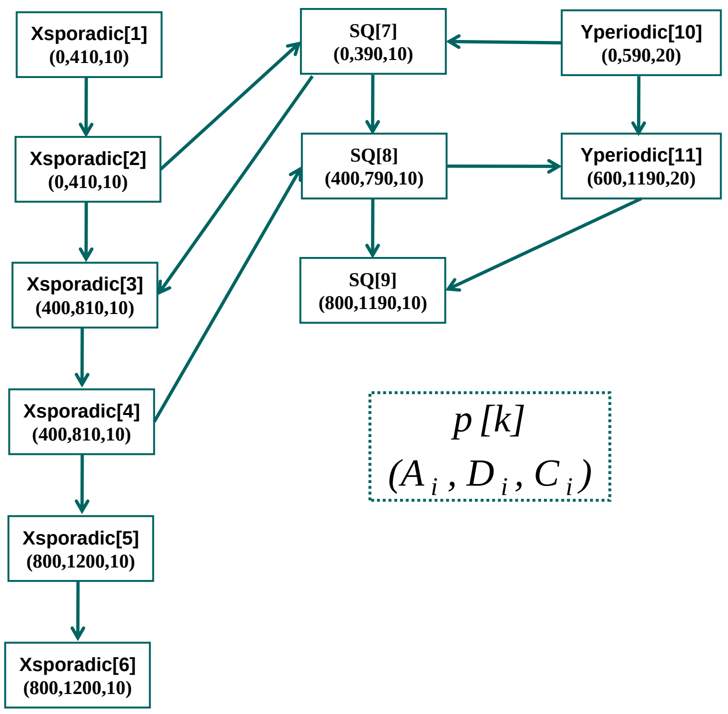


Figure 9: Task Graph for the "sqexample"

- **src folder files**

```
sq.c
sq.h
temp_asn1.h
xsporadic.c
xsporadic.h
xtrigger.c
xtrigger.h
yperiodic.c
yperiodic.h
```

CHAPTER 4. Scheduler, DOL, BIP Tools

Our real-time scheduler tool [RefSched] is intended to work in a single flow with DOL and BIP.

4.1 Offline Scheduler Tool

Note that the scheduler tool currently supports at most two criticality levels: “E” and “D”, where the task graph has two columns for the execution time (in normal and emergency mode – see task graph listing for “sqexample” earlier).

Below we show the fragment from the “**run**” script in which the scheduler tools are called:

```
BIP_RTE_WCET_DELTA=1000
NUM_PROC=3
SCHED_OUT=SCHEDULE_OUTPUT

$SCHED_TOOL fppn_tg.jobs $BIP_RTE_WCET_DELTA $NUM_PROC $SCHED_OUT
```

The scheduler tool is a script which calls some binary executables:

```
exe/scheduler/runSchedTool.sh
```

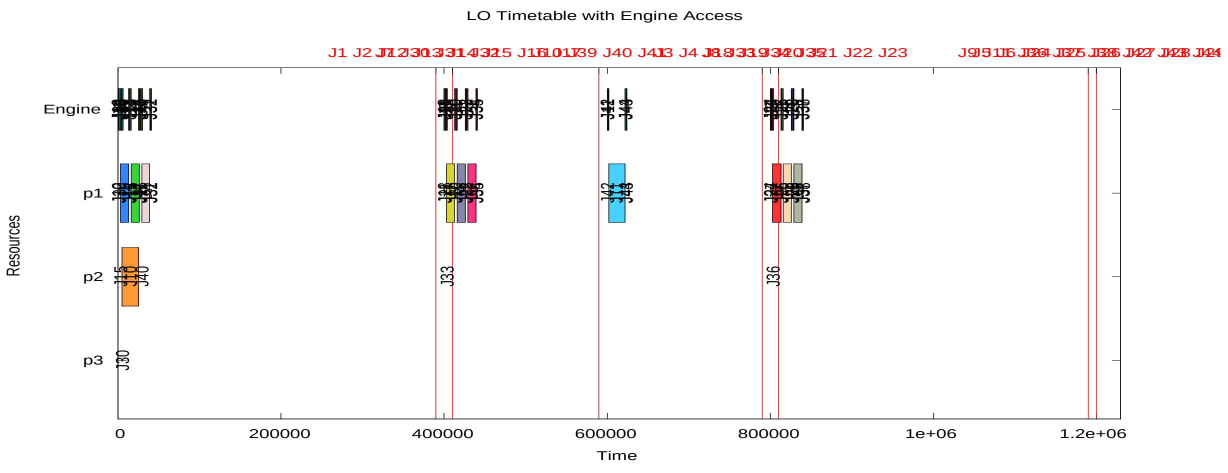
Next to the task graph we provide to the tool the value of `δ' – the WCET of one control transition in BIP RTE, the number of ***compute cores*** and the name mask for the output files. Note that the BIP RTE uses one core as ***control core***, where BIP RTE engine together with controller components run. So in this example in total 4 cores are used.

The script first scales the instance to obtain microseconds from milliseconds, then inserts extra nodes that model the task-controller transitions and then runs the scheduling algorithm (described in Technical Note on scheduling [RefSched]). The scheduler produces the LOG file where it reports some utilization metrics. For the “sqexample” we get:

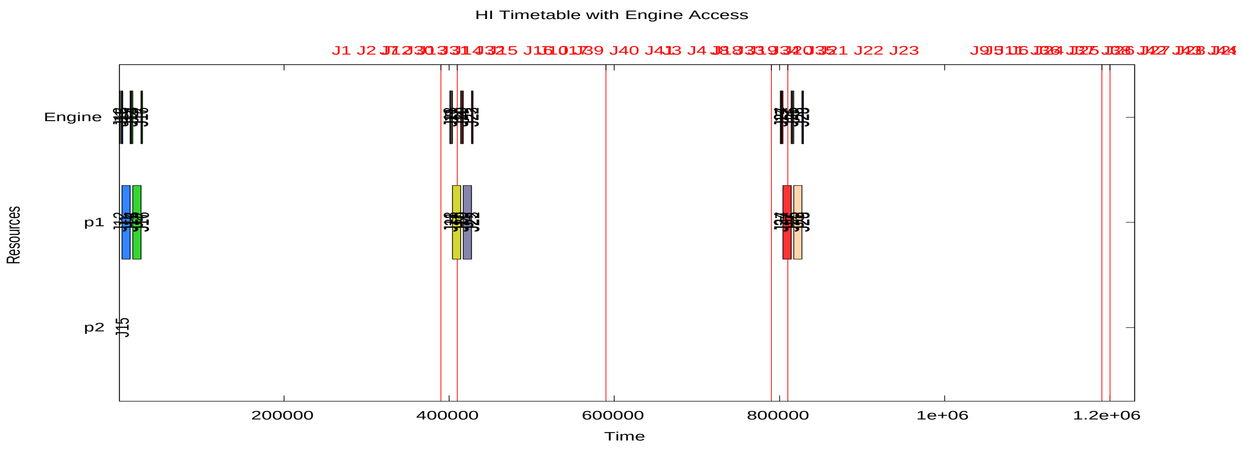
```
Utilization info:
Load Engine: 0.0366667
Load LO:     0.145028
Load HI:     0.070015
Mixed-mode utilization:
Load Engine: 0.0366667
Load MIX:    0.145028
```

Load LO and Load MIX give estimations of system load in `normal' mode (denoted as `LO' mode here). It is in this case around 14.5 %, based on the WCET times provided via Taste-IV. In emergency mode load is denoted as Load HI. The engine load is a metric on the load on the control core.

For the “sqexample” task graph the scheduler tool produces the Gantt charts SCHEDULE_OUTPUT_LO.pdf and SCHEDULE_OUTPUT_HI.pdf which correspond to time-triggered tables for normal and emergency modes.



sqexample: normal mode schedule



sqexample: emergency mode schedule

We see in both schedules a schedule for the control core (the engine) and for the compute cores (p1,p2,p3). These schedules are in fact the time-triggered tables for the two modes assuming single time triggered table per mode (STTM) scheduling policy. In the emergency mode only highly critical (level “D”) task is running (task “Xsporadic” in this example) and it is mapped to the same core as in the normal mode.

The offline scheduler tool assumes task migration, so it maps individual jobs but not tasks to the cores. Online scheduler BIP model does not yet support this feature, as BIP RTE engine needs some further development extensions for this. As we will see in later sections, the user has to provide the task-to-core mapping manually.

Currently the scheduler tools are prototype tools, their outputs are only Gantt charts. It is recognized that the user should be given possibility to improve the visual quality of the output. Most importantly, the scheduling tables produced by the offline scheduler should be integrated with online scheduling tools.

4.1.1. Scheduler Tools Example: “parallelproc”

One example is especially meant for the scheduler tool and for real-time execution on NGMP platform. It is called “parallelproc”. This example is described in Technical Note on scheduling [RefSched3.4] in detail as “Three Tasks Example”. One can find, for example, the files showing how ordinary scheduling example can be “modified” into mixed-critical one.

4.2 DOL Critical Frontend

DOL Critical language and tool suite was developed at ETH Zurich in collaboration with VERIMAG. A separate user manual is available [DOLC2BIP-Manual]. A concise introduction to DOLC can be found in [DOLC-BIP-Theory].

Here we refer to DOL Critical simply as **DOL**, but it should not be confused with the previous “non-critical” version developed in earlier projects. We provide DOL2BIP frontend for multicore code generation. This frontend translates the DOL programs into the BIP code that is ready as input for multi-thread code generation.

```
exe/dolc2bip/dolcbip.jar
```

Note that in DOL, the tasks are called *processes*, so we use terms “tasks” and “process” interchangeably in this document.

4.2.1. DOL Language Support in DOLC2BIP

The DOL frontend in the BIP tools support a subset of the DOL language.

Supported DOLC API functions for access to data channels from the processes:

- DOLC_read
- DOLC_write

DOLC also specifies some control-event communication, not supported in DOL2BIP conversion.

BIP tools provide some DOL extensions:

- aperiodic “protocol”

to specify how sporadic processes are called, e.g. *xtrigger.c* in ***sqexample***. Two additional parameters are supported in the application XML:

```
protocol_period
protocol_source
```

The period is the minimal distance between the events in aperiodic burst. The protocol source points to a user-defined C source file that should contain a function that “decides” whether the process should be activated. This function is called periodically with the period specified as “protocol period”.

- precedence constraints between *any* two processes

For example, in *sqexample* we have seen these constraints in the Xml file listing:

```
<!--Global requirements-->
<global name="globalCycle">
            <precedence chain="sq,xsporadic"/>
            <precedence chain="yperiodic,sq"/>
            </global>
</app>
```

In BIP tools for DOL, the precedence constraints can be enforced, using DOL syntax, between any two (a) periodic processes with any period.

The user should put a precedence constraint between every two *communicating* processes, from the higher-priority process to the lower-priority process. If the processes do not communicate then the priorities do not matter for their functional behaviors. For TASTE programs, this requirement is ensured automatically by the tools according to FPPN functional priority attribute.

In addition to the default use scenario for precedence constraints, we believe that they can be also used at the phase of porting single-core real-time applications to multi-cores. The precedence constraints can reproduce the "rate-monotonic" or any other fixed-priority assignment settings, popular in hard-real-time software designs.

Since now the offline scheduler tables are not yet integrated into BIP models, the user can use additional precedence constraints on the processes mapped to the same core to imitate fixed priority scheduling on the cores as a temporary replacement of automatically generated schedules.

4.3 Multi-Core BIP Programming for Timing-critical Applications

DOLC2BIP translator uses an extension of real-time BIP language to task automata, i.e. timed automata that control the scheduling of *tasks with self-timed execution* [DOLC-BIP-Theory].

4.3.1. Multi-threaded BIP Execution on Embedded Multi-core Platforms

The DOL-C code obtained from taste2dolc is translated into combined C/C++/Timed Automata variant of language BIP and then multi-core code is produced by translation from BIP to C++. The latter step is done by BIP compiler: **`bipc`** from the multi-thread distribution of BIP and requires the provided multi-thread library for BIP runtime. The multi-thread shared-memory model is assumed, implemented using POSIX multi-threading.

For multi-core BIP tools, see in the tools distribution:

```
exe/multi-bip/bin/bipc
exe/multi-bip/lib
exe/multi-bip/include
```

We have so far tried the tools for the following target platforms:

- standard Linux 32/64-bit
  - note: the 32-bit library provided by default is compiled for the Debian Linux of the 32-bit virtual machine provided
- LEON4™ NGMP from Cobham Gaisler with RTEMS SMP
  - This platform was temporarily available for us via MoSaTT-CMP project
- MPPA™ platform from Kalray with NodeOS™
  - This is a commercial platform provided by Kalray (www.kalray.eu)

The first target platform – Linux – is considered only for the purpose of prototyping, as we see no way to guarantee predictable real-time execution in this case. Still, this option works for simple examples (see the **`launch`** script in our examples). The libraries for other multi-core platforms could be compiled on special request, the main requirement is support of C++ compilation, Posix Threads and mapping of threads to SMP cores.

For the BIP language we use its *real-time* variant, which supports timed automata clocks (www.bip-components.com), with some important deviations:

- only rendezvous interactions are supported (e.g. no `broadcast' interactions)
- BIP priorities are not supported
- timing constraints for internal component transitions are not supported. Their semantics is different: they may take *any* time to execute ("*self-timed*" transitions), as opposed to default instantaneous transitions (note, this is a new experimental feature).

4.3.2. Thread-map file

Currently BIP RTE assumes one-to-one mapping on threads to cores. Therefore we use terms thread and core as synonyms here. The user or Taste2BIP flow provides a textual file to BIP compiler (via command line option) called *thread-map* file. This file specifies how BIP Components should be mapped to threads (cores).

There is one reserved thread name "bipthread"; we call it *control core* sometimes. This is the thread where the BIP RTE engine is running. The user is supposed to map to that thread all BIP components that are responsible for control, so basically all BIP components in the design. Exceptions are BIP components responsible for processes (tasks). They are mapped to threads with different arbitrary names, those threads will run on "compute cores". Below we give an example of thread-map file for "sqexample":

```
bipthread Controller_xsporadic
bipthread cPeriodicSourcesq
bipthread cRelDlSinksq
bipthread cPeriodicSourceyperiodic
bipthread cRelDlSinkyperiodic
bipthread xmailbox_ins
bipthread yblackboard_ins
bipthread Prec_sq_xsporadic
bipthread Prec_yperiodic_sq
core1 xsporadic_ins
core2 sq_ins
core3 yperiodic_ins
```

Of course, we could have mapped all three processes to the same core, the above thread-map is just an example.

Conceptually, this file should be generated by the offline scheduler tool. However, the current offline scheduler assumes that processes (i.e. tasks) can run jobs at different cores (in other words, can migrate from one core to another), whereas thread map file assigns a single core (i.e. thread) per process (i.e. tasks). For example, in our thread-map example above the last three BIP components, shown in red color, which correspond to the three processes, are mapped to three different cores.

Currently the DOLC2BIP tool can generate a default thread-map file (via --thread_map_output command line option, see “run” script of “parallelproc”example). In this default file all BIP components are mapped to “bipthread”. Then the user can modify this file to map the process components to different cores, like in the example above. A BIP component corresponding to a process can be recognized by its name, which is “process name” followed by “_ins”.

4.3.3. Manual modifications in main.cpp file and Hidden Channels

When building for embedded multi-core system the “main.cpp” file of BIP RTE is provided separately, so that the user can modify it, to properly configure the settings of Embedded RTOS, such as RTEMS for LEON4 platform and NodeOS for MPPA. The typical configuration modifications include setting the maximal stack size per thread and number of resources of different type (threads, semaphores etc.)

If an example is supposed to run for real-time on multi-core, for better performance it is required to “hide” (i.e. set a Boolean attribute “connector is hidden”) all connectors between the BIP components that represent processes (tasks) on one side and data channels (mailboxes and blackbords) on the other. “Hiding” means that the given connectors are not coordinated in a centralized way no by BIP engine but instead are coordinated by the process itself in a distributed way. In a hidden connector one port is called “master” the others are “slaves”. In process- channel connectors, the process port is marked as master and the channel ports as slaves. Even if the data channel is mapped to the “bipthread”, when a slave port is executed, all the associated operations and coordination is done in the thread of the master, without involving BIP RTE engine, which makes an important difference in performance.

In the older versions of the tool this had to be done manually, via main.cpp file, in the current version this is done automatically. The older version style can be found in “parallelproc” and “sqexample”, the newer version can be found in “producer_consumer” example.

In the older version, one had to fill-in function markPorts in “main.cpp”. Below we give an example for “parallelproc” benchmark, where there are two mailbox components.

“Old-style” channel hiding: manually filling in a function in mai.cpp: “parallelproc” example.

```
void markPorts(Compound* system) {
    markMBconnectorsMaster (system, "split",   "p_PROC_XIF_Write2",       "processa",
"p_PROC_XIF_Read2",  "processa");
    markMBconnectorsMaster (system, "split",   "p_PROC_XIF_Write",        "processb",
"p_PROC_XIF_Read",   "processb");
    markMBconnectorsSlave (system, "xmailbox");
    markMBconnectorsSlave (system, "xmailbox2");
}
```

In the current version, it is enough to add “--hidden_ports_output” option to DOL2BIP tool. The tool will generate a textual file that configures the bipc compiler (via its command line option) to generate the above manual code automatically. See the `launch` script in “producer_consumer” example.

4.3.4. Obtaining Platform Measurement Gantt Chart

The execution of FPPN processes and BIP RTE engines should be instrumented for timing measurements. The provided build of BIP RTE engine assumes that the reader implements the functions declared in gantt.h in BIP RTE lib.

```
void mark_start(int id, int core);
void mark_end(int core);
void print_gantt(int niter);
void chart_init();
void reset_gantt_time();
void count_inter();
```

chart_init() and **reset_gantttime**() are called by BIP RTE at the beginning of execution, at the time instant "zero".

mark_start (id=20, core=0) is called at the beginning of a series of BIP RTE engine steps and **mark_end**(core=0) is called at the end. The core argument should be ignored, and "id" is "process id", 20 is reserved for BIP engine. The user processes can call these two functions at the start and at the end of firing with his unique id for instrumentation. **count_iter**() is used to count engine steps (BIP control-automata transitions) inside the block between start and end.

See "parallelproc" example for use and implementation of these functions. Important parameter in this implementation is "STOP_NUM" that measures the number of engine steps before simulation is stopped. It determines the length of the Gantt chart produced.

This parameter is set in ***global.c file in "parallelproc" example***:

```
……
#define P_NUM 10
#define ENTRY_NUM 256
#define STOP_NUM 50
typedef struct chart{
    int start[P_NUM][ENTRY_NUM];
    int finish[P_NUM][ENTRY_NUM];
    int id[P_NUM][ENTRY_NUM];
    int curr[P_NUM];
    int overhead;
}Chart;
Chart gantt;
int interNum[ENTRY_NUM];


void chart_init(){
….
```

The implementation of Gantt chart prints at the end to the standard output the textual trace of measurements, for example for "parallelprocs":

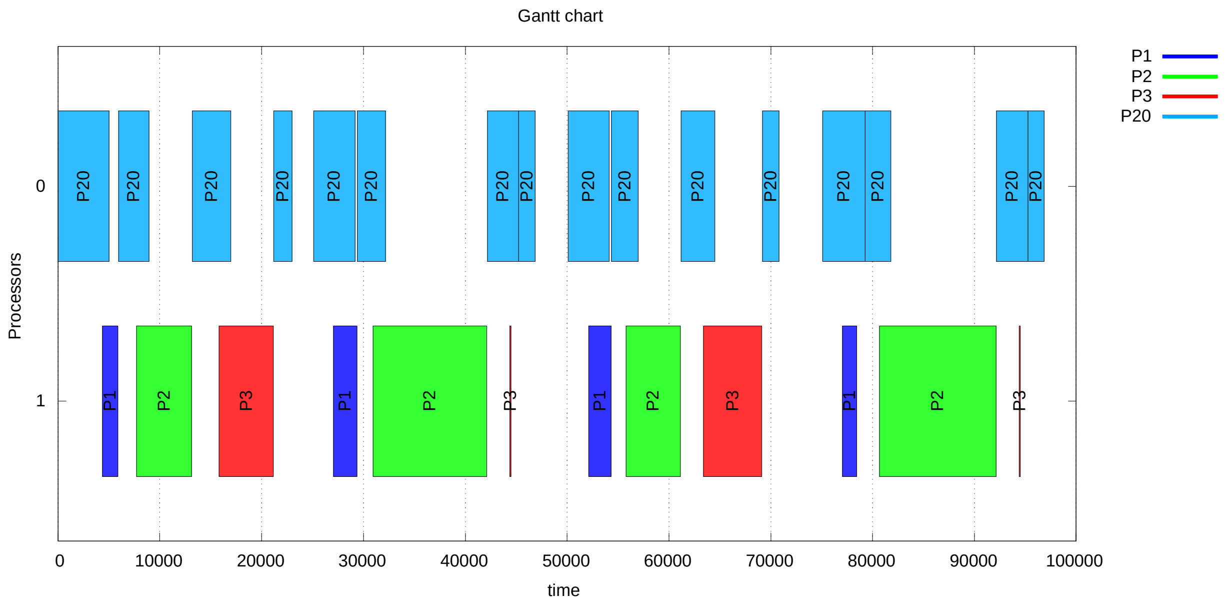
```
0       4       5012    P20 - ET=5008 NumItnter=4
0       5934    8915    P20 - ET=2981 NumItnter=3
0       13181   16960   P20 - ET=3779 NumItnter=4
0       21197   22968   P20 - ET=1771 NumItnter=2
0       25095   29162   P20 - ET=4067 NumItnter=4
0       29394   32181   P20 - ET=2787 NumItnter=3
0       42174   45230   P20 - ET=3056 NumItnter=4
0       45233   46860   P20 - ET=1627 NumItnter=2
0       50095   54152   P20 - ET=4057 NumItnter=4
0       54355   56964   P20 - ET=2609 NumItnter=3
```

```
0       61176   64524   P20 - ET=3348 NumItnter=4
0       69192   70811   P20 - ET=1619 NumItnter=2
0       75095   79267   P20 - ET=4172 NumItnter=4
0       79270   81798   P20 - ET=2528 NumItnter=3
0       92176   95246   P20 - ET=3070 NumItnter=4
0       95249   96849   P20 - ET=1600 NumItnter=2
1       4355    5876    P1 - ET=1521
1       7726    13126   P2 - ET=5400
1       15835   21142   P3 - ET=5307
1       27068   29340   P1 - ET=2272
1       30937   42119   P2 - ET=11182
1       44380   44502   P3 - ET=122
1       52109   54302   P1 - ET=2193
1       55785   61121   P2 - ET=5336
1       63410   69138   P3 - ET=5728
1       77056   78458   P1 - ET=1402
1       80687   92122   P2 - ET=11435
1       94408   94520   P3 - ET=112
```

This standard output can be saved to a text file, let us name it GANTT_OUT.txt. Then the following command in out tools will generate a graphical Gantt chart:

```
runMeasGantt.sh GANTT_OUT.txt
```

This command will ask the user confirmation for the generated gnuplot file (it will be required to check it in VIM editor and the exit from it by :q command). Then it will create GANTT_OUT.pdf, like shown below for the above example:



References

[RefSched] P. Poplavko, R. Kahil, D. Socci, S. Bensalem, M. Bozga. *Technical Note.* **Ensuring Schedulability for Embedded Multi-cores.**

[RefUseCase] F. Gioulekas, P. Poplavko, A. Zerzelidis, P. Katsaros, P. Palomo. *Technical Note*. **Application Use Case for Multi-core Schedulability**.

[DOLC2BIP-Manual] P. Poplavko, P. Bourgos, M. Bozga, S. Bensalem. *Multicore Code Generation for Time-critical Applications using BIP Tools*: (**DOLC-to-BIP Manual**).

[DOLC-BIP-Theory] G. Giannopoulou, P. Poplavko, *et al*, **DOL-BIP-Critical: A Tool Chain for Rigorous Design and Implementation of Mixed-Criticality Multi-Core Systems**. TIK Report No. 363, ETH, Zurich, 2016.