

# A Tutorial on TTool/DIPLODOCUS: an Open-source Toolkit for the Design of Data-flow Embedded Systems

Andrea Enrici<sup>1</sup>, Letitia Li<sup>2</sup>, Ludovic Apvrille<sup>2</sup>, and Dominique Blouin<sup>2</sup>

<sup>1</sup> Nokia Bell Labs France

Centre de Villarceaux, Route de Villejust 91620, Nozay, France

[andrea.enrici@nokia.com](mailto:andrea.enrici@nokia.com),

<sup>2</sup> LTCI, CNRS, Telecom ParisTech, Université Paris-Saclay, 75013, Paris, France  
[{letitia.li, ludovic.apvrille, dominique.blouin}@telecom-paristech.fr](mailto:{letitia.li, ludovic.apvrille, dominique.blouin}@telecom-paristech.fr)

**Abstract.** TTool/DIPLODOCUS is an open-source Computer Aided Design tool that allows the modeling, verification (simulation and formal verification) and automatic code generation of data-flow embedded systems, from UML/SysML diagrams.

This document is a tutorial for TTool/DIPLODOCUS that guides the reader through the complete design of the data-link layer of a ZigBee transmitter [6]. We first provide a short introduction that motivates the use of Electronic Design Automation and Model-Driven Engineering to facilitate the programming of modern embedded systems. Subsequently, we present a generic overview of TTool/DIPLODOCUS (Section 3) and of its software architecture (Section 4). Section 5 guides the user through the installation and configuration process. This is followed by the tutorial that illustrates the full design of the ZigBee transmitter, from modeling (Section 7) to Design Space Exploration (Section 8) and automatic code generation (Section 9). Section 11 concludes this document.



# Table of Contents

A Tutorial on TTool/DIPLODOCUS: an Open-source Toolkit for the Design of Data-flow Embedded Systems . . . . .	1
<i>Andrea Enrici, Letitia Li, Ludovic Apvrille, and Dominique Blouin</i>	
1 Important note . . . . .	3
2 Why TTool/DIPLODOCUS? . . . . .	4
3 An overview of TTool/DIPLODOCUS . . . . .	4
4 The software architecture of TTool/DIPLODOCUS . . . . .	6
5 Configuring TTool/DIPLODOCUS . . . . .	6
6 Starting a new project . . . . .	10
7 Modeling a ZigBee transmitter . . . . .	15
7.1 The functionality of a ZigBee transmitter (data-link layer) . . . . .	15
7.2 Creating the application model of a ZigBee transmitter (data-link layer) . . . . .	16
Attributes of a primitive component . . . . .	19
Ports, channels, events and requests . . . . .	19
The activity diagram of a primitive component . . . . .	22
7.3 Platform modeling . . . . .	32
7.4 Creating the platform model of EMBB . . . . .	32
7.5 Communication protocols and patterns modeling . . . . .	40
7.6 Modeling a DMA data transfer with Communication Patterns . . . . .	41
7.7 Communication models . . . . .	43
The communication mismatch in EMBB . . . . .	45
Creating Communication Pattern diagrams . . . . .	50
7.8 Mapping . . . . .	53
8 Design Space Exploration in TTool/DIPLODOCUS . . . . .	66
8.1 Simulation . . . . .	66
The simulation results of the ZigBee transmitter (physical layer) . . . . .	72
8.2 Formal verification . . . . .	77
Formal Verification before mapping . . . . .	77
Pre-mapping formal verification with UPPAAL . . . . .	79
Post-mapping formal Verification with the TTool verifier and simulator engine . . . . .	79
Example of post-mapping formal verification . . . . .	80
Post-mapping formal verification with ProVerif . . . . .	83
9 Automatic Code Generation for Rapid Prototyping . . . . .	86
The compilation process . . . . .	86
Scheduling of operations . . . . .	87
Memory allocation . . . . .	87
Portability of the code-generation approach . . . . .	87
9.1 Generating the code for the ZigBee transmitter . . . . .	88
10 Analysis of security properties . . . . .	93
10.1 Symmetric Encryption . . . . .	93
10.2 Nonces . . . . .	94
10.3 Key exchange . . . . .	94
10.4 MAC . . . . .	95
10.5 Automated Security Generation . . . . .	95
11 Conclusion . . . . .	98
1.A Formal description of Communication Patterns . . . . .	99
1.B TTool/DIPLODOCUS' simulation semantics . . . . .	102
1.B.1 Functionality . . . . .	102
1.B.2 Platform . . . . .	102
1.B.3 Mapping . . . . .	103

## 1 Important note

The screen capture provided in this tutorial have been made with a former version of TTool. The graphical interface of the version of TTool you will use to follow this tutorial may thus differ, e.g. you will have to find the corresponding icon in your version. To do so, you can place your mouse over an icon to get an information on it. By doing so, you should be able to locate when icons have been moved.

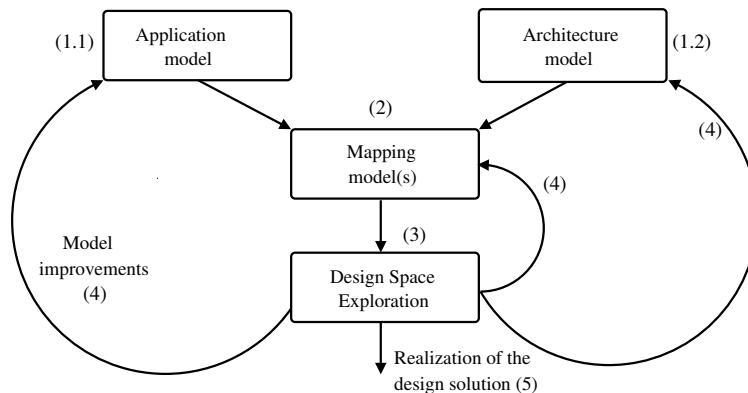
## 2 Why TTool/DIPLODOCUS?

To provide more computational power, modern embedded systems are more and more realized as parallel systems where the processing and control are distributed over a network of interconnected subsystems. This type of architecture is used for data-flow applications where performance is driven by the need to rapidly process and transfer large volumes of data (e.g., signal, video and image processing). Currently, we find that parallel and distributed architectures are largely adopted both at the chip level (e.g., Multi-Processors Systems on Chip) or in domains where the electronic components are physically distributed over the structure of the entire system (e.g., automotive and avionics domains). These architectures are intrinsically difficult to program because of their high degree of parallelism, the complex interactions between hardware and software components, and their heterogeneous nature. They are typically assembled with components from different vendors (e.g., CPU, RAM motherboard and its communication infrastructure) and with different characteristics (e.g., memory architecture, Application Programming Interface).

In this context, an important challenge is to efficiently program and verify these complex platforms, and reduce the time-to-market of new products, development time and costs.

Among the possible approaches that can be taken to alleviate software development, Model Driven Engineering [1] proposes to raise the level of abstraction at which these systems are programmed. Instead of directly encoding a given functionality (e.g., a signal-processing algorithm such as LTE) into code that will be executed by a target platform, MDE aims to capture this functionality in high-level models that are independent of a specific target platform or implementation technology. These models are then coupled with transformation engines that can automatically generate artifacts (e.g., code, documentation) for multiple targets and implementations.

To design an embedded system, traditionally an engineer separately models both the application(s) - i.e., the functional part of the system - and the candidate platform - i.e., the hardware/software resources - with the assistance of dedicated Electronic Design Automation (EDA) tools. Then he/she selects (*maps*) the platform units that will execute the function's workload. The resulting design, given by the mapping model, is evaluated in terms of performance (e.g., power consumption, throughput, latency, silicon area occupation) and compared to a set of requirements. In case of a positive match, an implementation (i.e, compilable software, synthesizable hardware) is generated via automatic model transformations. In case of a negative match, the application and platform models are iteratively improved, mapped and evaluated until the resulting design complies with the desired requirements. This design process is known as the Y-chart approach [5] and is shown in Fig. 1.

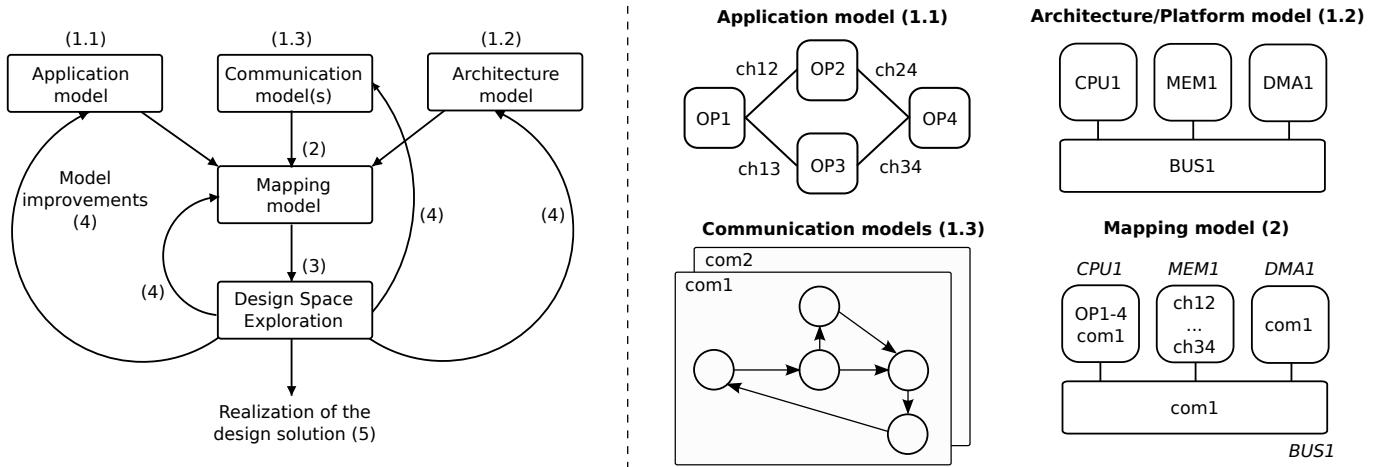


**Fig. 1.** The Y-chart approach for the design of programmable embedded systems

## 3 An overview of TTool/DIPLODOCUS

TTool/DIPLODOCUS is an open-source EDA toolkit dedicated to the design of data-flow embedded systems based on UML and SysML diagrams. To go into more detail, TTool is the name of the toolkit and DIPLODOCUS is the name of a specific UML/SysML profile dedicated to the partitioning of embedded systems. TTool also supports the AVATAR, TURTLE, SysML-Sec, CTTool and Network Calculus profiles [2]. TTool/DIPLODOCUS [7], [8] [2] uses an evolution of the Y-chart approach that is called the  $\Psi$ -chart approach, shown in Fig. 2. In the  $\Psi$ -chart approach, a third set of input models is introduced to capture communication protocols and patterns (step 1.3 in Fig. 2) independently of the application and platform models. This has proven to reduce the design time in terms of the number of iterations in

the Model improvement phase (step 4 in Fig. 1 and in Fig. 2) as well as to improve the portability of models [18] to target multiple platforms.



**Fig. 2.** The  $\Psi$ -chart approach (left side) and a graphical visualization of its constituent models (right side).

In TTool/DIPLODOCUS, an *application* model (step 1.1 in Fig. 2) is denoted with SysML Block Definition and Activity diagrams. An application is described as a set of blocks interconnected by data and control dependencies via ports and channels. The internal behavior of each block is described by a SysML Activity Diagram. An application model is based on the two following abstraction principles:

- *Data abstraction*: only the amount of data exchanged between application blocks is modeled. Internal decisions that depend on the value of data are expressed in terms of non-deterministic and static operators (i.e., conditional choice based on the value of a random variable).
- *Functional abstraction*: algorithms are described using abstract cost operators that express the complexity of processing data in terms of the number of operations required to execute them (e.g., number of integer operations).

A *platform* model (step 1.2 in Fig. 2) is denoted using a UML Deployment Diagram that represents a set of generic interconnected resources, e.g., bus, CPU and its operating system, DMA, memory. These resources are characterized by performance and implementation parameters (e.g., the scheduling policy and the number of cores for a CPU, the addresses of memory areas) that are used for Design Space Exploration (DSE), evaluation (e.g., simulation, formal verification), and to realize a design solution (i.e., control code synthesis). A *communication* model (step 1.3 in Fig. 2) is denoted using SysML Activity Diagrams and UML Sequence Diagrams that represent, respectively, the algorithm of a communication protocol and the entities (e.g., master, slave) that are involved in exchanging information. A *mapping* model (step 2 in Fig. 2) is created from an instance of the platform model where dedicated UML artifacts are added to map computations and communications. The abstract cost operators are assigned a value according to the performance characteristics (e.g., operating frequency) of the platform's units. TTool/DIPLODOCUS allows a user to map functions that belong to different functional views (i.e., application models) and to different communication models.

Design Space Exploration in TTool/DIPLODOCUS (step 3 in Fig. 2) evaluates the performance of a mapping solution by simulating the workload of computations and data-transfers [4]. A formal verification engine [4] is also available to verify system properties (e.g., liveness, reachability, scheduling). DSE can be performed both manually via the tool's GUI or automatically via a set of scripts that configure the DSE engine to evaluate different mapping alternatives.

The realization of a design solution in TTool/DIPLODOCUS (step 5 in Fig. 2) is possible in terms of a software implementation, via the automatic generation of control code. This code corresponds to the functionality of the application and communication models as mapped onto dedicated resources of the platform model. From a software viewpoint, this control code is an application that runs on top of the Operating System of a general-purpose processor in the target platform.

The abstraction principles underlying models of TTool/DIPLODOCUS answer the need to target early design and DSE,

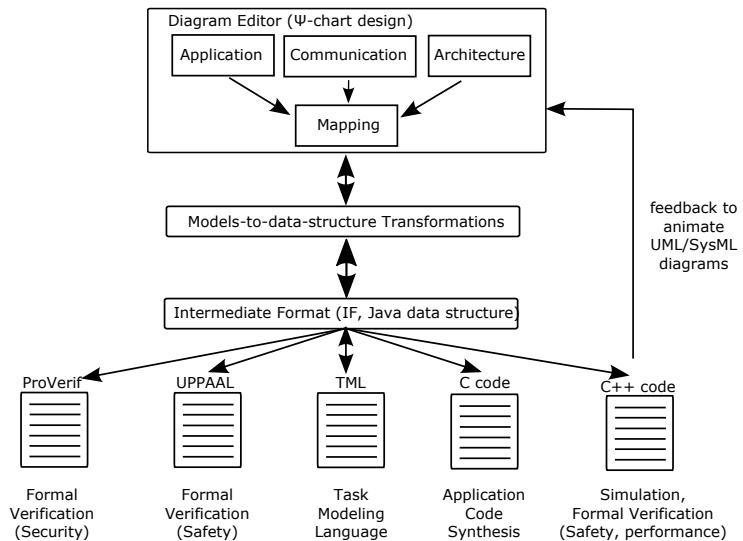
when not all the details about a system's application (e.g., value and type of data) and platform (e.g., Operating System, size and policy of cache memories for a CPU) are known. The validation of the effectiveness of these abstractions has been described in [10], where TTool/DIPLODOCUS was used for the design of the physical layer of a LTE base station jointly with Freescale Semiconductors. The resulting design in TTool/DIPLODOCUS lead to performance results that differed by only 10% with respect to the final implementation. To obtain these performance figures, design in TTool/-DIPLODOCUS required only a few weeks, whereas manual development of a functionally equivalent system amounted to 6 months.

## 4 The software architecture of TTool/DIPLODOCUS

Fig. 3 illustrates the software architecture of TTool that is relevant to the DIPLODOCUS profile. Most of the software that composes TTool is developed in Java, except for the Design Space Exploration engine that has been developed in C++ for performance reasons.

The topmost level of Fig. 3 shows the Diagram Editor, an in-house Java Graphical User Interface that permits designers to draw UML/SysML diagrams. Graphical models are first converted into an Intermediate Format (IF) Java data structure (Models-to-data-structure Transformation, Intermediate Format in Fig. 3). This software data structure constitutes a common layer from which models are transformed for DSE via formal verification, simulation and for implementation via the synthesis of executable control code.

TTool/DIPLODOCUS also accepts as input a textual representation of a design, specified in a dedicated language called Task Modeling Language (TML) [9]. This TML representation, Fig. 3, is manually input by the user, as an alternative to UML/SysML diagrams. Instead, when the latter are used, a TML description can be automatically produced from the IF data structure, Fig. 3.



**Fig. 3.** The software architecture of TTool for the UML/SysML profile DIPLODOCUS

## 5 Configuring TTool/DIPLODOCUS

In this section, we describe the steps that are necessary to install and configure TTool/DIPLODOCUS in a Linux environment.

The source files of TTool can be downloaded in form of a .tgz archive from the website <http://ttool.telecom-paristech.fr/download.html>, after having accepted the licence agreement. To install the tool, simply unpack the archive in your home directory:

```
$ cd
$ tar -xzvf releaseTToolWithSrc_RELEASENUMBER.tgz
```

A folder named TTool will be created in your home directory. A preconfigured version of the tool can be downloaded from the website and a version of TTool inside a Virtual Machine is also available upon request to [ludovic.aprille@telecom-paristech.fr](mailto:ludovic.aprille@telecom-paristech.fr). In case the user wants to customize the configuration of TTool, a dedicated configuration file, `config.xml` located in `your-home/TTool/bin/>`, must be modified as follows:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE TURTLECONFIGURATION>
<TURTLEHost data="localhost" />
<RTUHost data="your-local-path-to rt1" />
<RTUPath data="your-local-path-to rt1" />
<DDADOPPath data="your-local-path-to rr2tisa" />
<RG2TSAPath data="your-local-path-to rstrrap" />
<RG2TRAPPPath data="your-local-path-to rstrrap" />
<DOTTYPath data="your-local-path-to dotty" />
<DOTTYHost data="localhost" />
<AldebaranPath data="your-local-path-to aldebaran" />
<AldebaranPath data="your-local-path-to aldebaran" />
<BegOfPath data="your-local-path-to beg->10" />
<BeginPath data="your-local-path-to bkg->min" />
<BeginPath data="your-local-path-to bkg->open" />
<BisimulationPath data="your-local-path-to bkg->merge" />
<BisimulationPath data="your-local-path-to bkg->caesar" />
<CaesarPath data="your-local-path-to caesar" />
<CaesarOpenPath data="your-local-path-to caesar" />
<CaesarOpenPath data="your-local-path-to caesar" />
<FILEPath data="your-home/TTool/modeling" />
<LIBPath data="your-home/TTool/lib" />
<IMGPath data="your-home/TTool/figure" />
<LOTOSPPath data="your-home/TTool/lotos" />
<GGraphPath data="your-home/TTool/graphs" />
<TToolUpdateURL data="your-home/TTool/modeling" />
<TToolUpdateProxyPort data="false" />
<TToolUpdateProxyPort data="8080" />
<TToolUpdateProxyPort data="To Be Completed" />
<JavaCodeDirectory data="your-home/TTool/javacode" />
<JavaCompilerPath data="usr/bin/javac" />
<JavaCompilerPath data="your-home/TTool/javacode" />
<JavaClassPath data="your-home/TTool/tmicode" />
<JavaExecutePath data="usr/bin/java" />
<JavaHeader data="import java.*;" />
<SystemCCodeDirectory data="your-home/TTool/ccode" />
<SystemCHost data="localhost" />
<SystemCodeCompileCommand data="make -C your-home/TTool/simulators/c++2/r" />
<SystemCodeExecuteCommand data="make -C your-home/TTool/simulators/c++2/run.x -oed your-home/TTool/simulators/c++2/run.x -server" />
<SystemCodeInteractiveExecuteCommand data="your-home/TTool/tmicode" />
<TMLCodeDirectory data="your-home/TTool/tmicode" />
<CCodeDirectory data="/opt/local/bin/gtkwave" />
<GTRWavePath data="/opt/local/bin/gtkwave" />
<VCDFPath data="vcdf/ved/" />
<UPPAALCodeDirectory data="your-home/TTool/uppaal" />
<UPPAALVerifierPath data="your-home/TTool/uppaal/bin_Linux/verifty.a" />
<UPPAALVerifierPath data="your-home/TTool/uppaal/bin_Linux/verifty.a" />
<ProVerifCodeDirectory data="your-home/TTool/proverif" />
<ProVerifVerifierPath data="your-local-path-to proverif" />
<ProVerifVerifierHost data="localhost" />
<AVATARExecutableCodeDirectory data="your-home/TTool/executablecode" />
<AVATARMPSCCodeDirectory data="your-home/TTool/mpsoc" />
<AVATARMPSCCodeHost data="localhost" />
<AVATARMPSCCodeCompileCommand data="make -C your-home/TTool/executablecode compilesoclib" />
<AVATARMPSCCodeExecuteCommand data="make -C your-home/TTool/executablecode run.x" />
<AVATARMPSCCodeUpdateGeneratedCode data="your-home/TTool/mpsoc updategeneratedcode compilesoclib" />
<AVATARMPSCCodeUpdateGeneratedCode data="make -C your-home/TTool/mpsoc runsoclib" />
<AVATARMPSCCodeTraceCommand data="make -C your-home/TTool/mpsoc runsoclib-trace" />
<AVATARMPSCCodeTraceFile data="make -C your-home/TTool/mpsoc runsoclib-trace" />
<ExternalCommandHost data="localhost" />
<ExternalCommand data="an external-command-of-your-choice, e.g., /opt/local/bin/gtkwave your-local-path-to-a-simulation-trace" />
<ExternalCommandHost data="localhost" />
<LastOpenFile data="the-path-of-your-last-opened-TTool-file" />
<LastWindowAttributes x="1097" y="89" width="1075" height="848" max="false" />
<ProVerifHash data="" />
</TURTLECONFIGURATION>

```

TTool can also run on a Windows PC: a dedicated folder located under `TTool/preinstallTTool/windows` contains the executable file `ttool.bat` that directly launches the tool.

Once the configuration file has been modified, to start TTool, we recommend the user create an executable script called `ttool.exe` in the installation directory of TTool, `<your-home>/TTool/`:

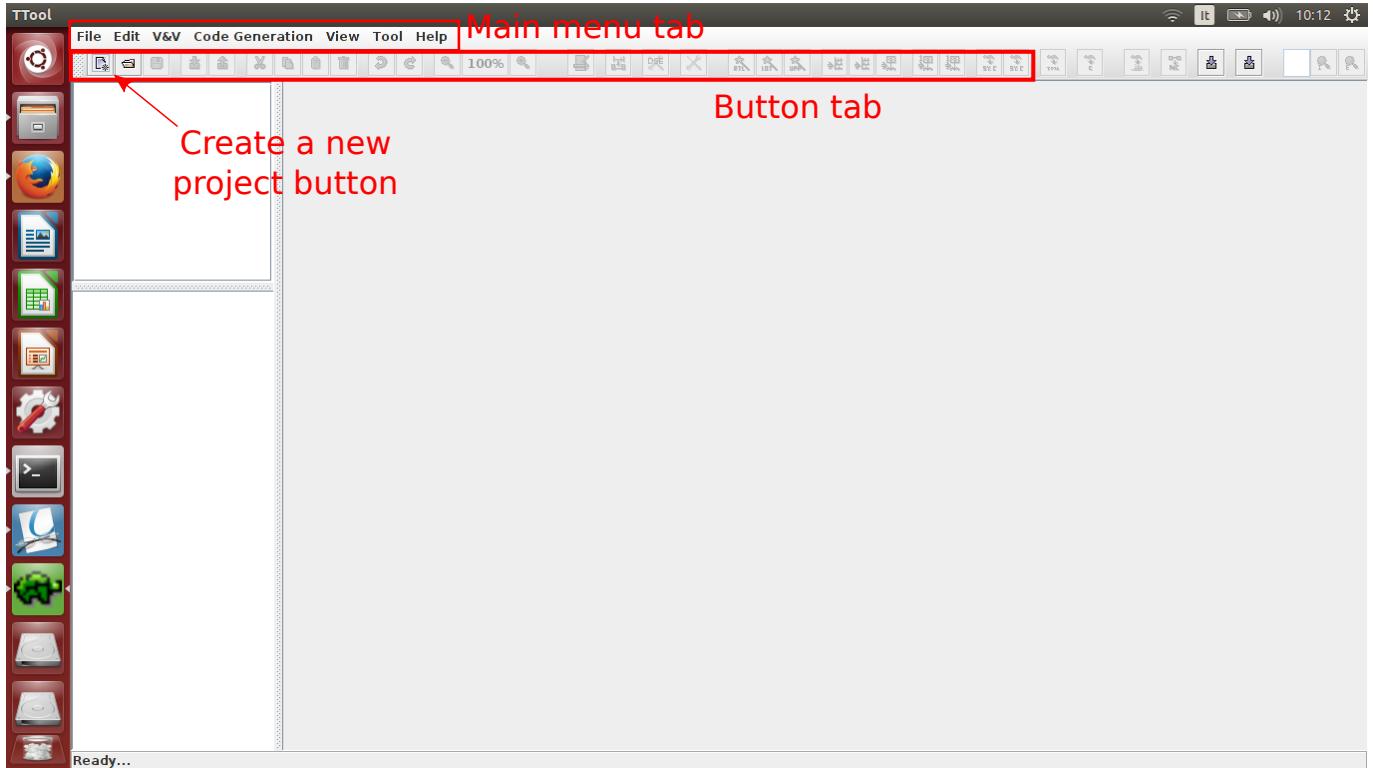
```
$ touch your-home/TTool/ttool.exe
$ echo "#! /bin/sh" > your-home/TTool/ttool.exe
$ echo "java -Xmx1024m -jar your-home/TTool/bin/ttool.jar -config your-home/TTool/bin/config.xml
-avatar uppaal -launcher" >> /TTool/ttool.exe
$ chmod u+x your-home/TTool/ttool.exe
```

Running the script file will launch the GUI of TTool:

```
$ ./your-home/TTool/ttool.exe
```

## 6 Starting a new project

To create a new project, click on the New icon shown in Fig. 4 or select File->New from the Main menu tab. For ease of use, many options from the main menu, such as the creation of a new project, have a corresponding button in the Button tab.



**Fig. 4.** Create a new project in TTool/DIPLODOCUS

Fig. 5 shows the main window of TTool/DIPLODOCUS as it appears after creating a new project. It is divided into three areas: the Project navigation window, the Map-view window and the Design window. The Project navigation window allows a user to navigate through the files of a project, to check for results of the syntax analysis and to rapidly search for elements of a design. The Design window is the window where the UML/SysML diagrams of a design will be displayed. The Map-view window shows a bird's eye view of the diagram that is currently displayed in the Design window.

Apart from the single application, communication, platform and mapping models (Fig. 2) that compose a design project, TTool/DIPLODOCUS also allows the creation of a Methodology diagram that represents the  $\Psi$ -chart approach. To create such a Methodology diagram, right click in the Design window and select **New DIPLODOCUS Methodology**, this will create the diagram shown in Fig. 6. For each box in Fig. 6, a reference to a diagram can be added by right-clicking on it and selecting **Add diagram reference**. This opens a dedicated window such as the one in Fig. 7, where, for instance, the references of up to 3 application diagrams can be added. After their addition, the Methodology diagram appears as in Fig. 8.

With respect to the terminology used so far, before continuing the tutorial, we now distinguish between a *panel* and a *diagram*. Generally speaking, a panel is a diagram container that can contain more than one diagram. In Fig. 9, we graphically show this difference with a screen capture of the complete ZigBee design. The DIPLODOCUS\_Methodology panel can contain only one diagram, whereas panels for other models (application, communication, mapping, platform) can contain more than one diagram.

At this point we can save the project by clicking on the dedicated button, Fig. 10 or by selecting **File->Save as** from the main menu bar. We will name the project **ZigBeeTX** as shown in Fig. 11 and save it in the folder that is purposed by default by the tool for all projects. This folder is located in **your-home/TTool/modeling/**. We now continue our tutorial with the description of the data-link layer of the ZigBee transmitter that we will design. Each

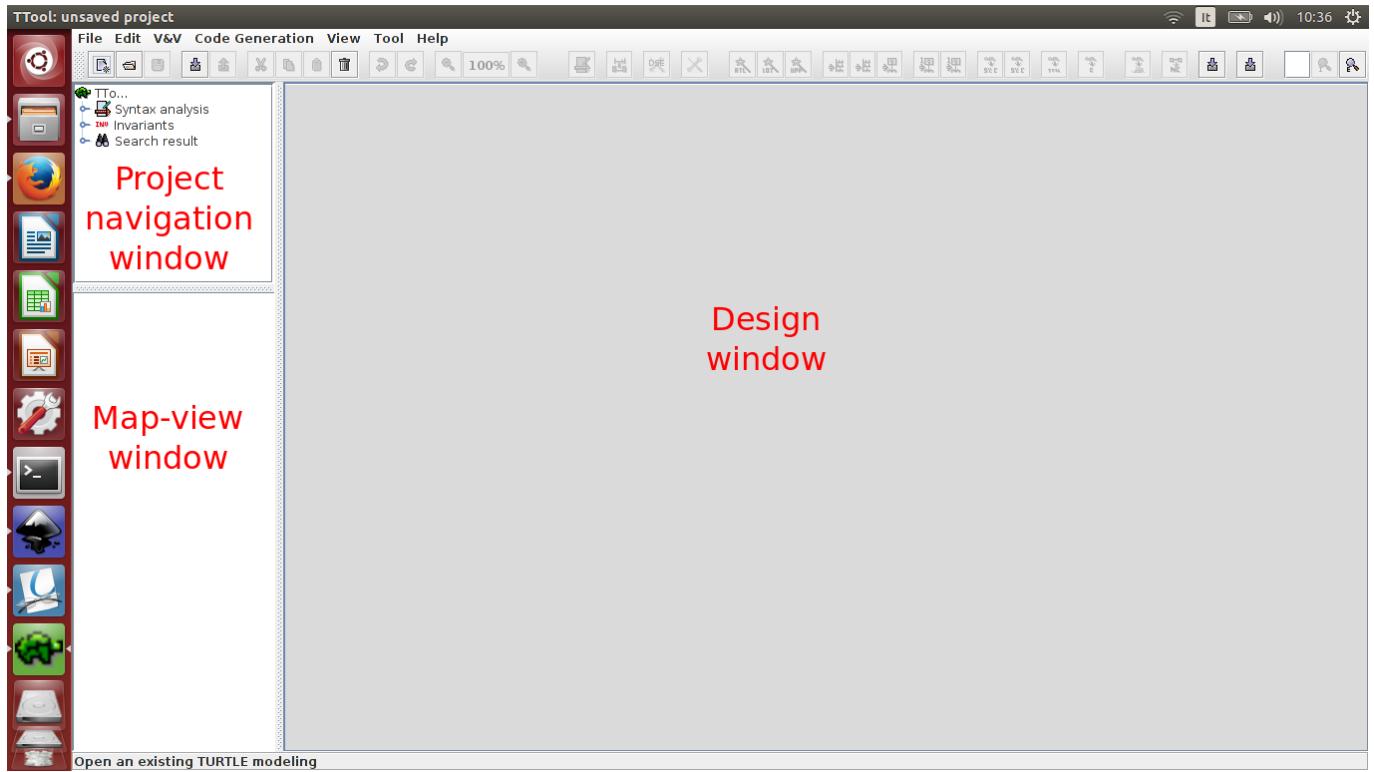


Fig. 5. The windows that compose a new project

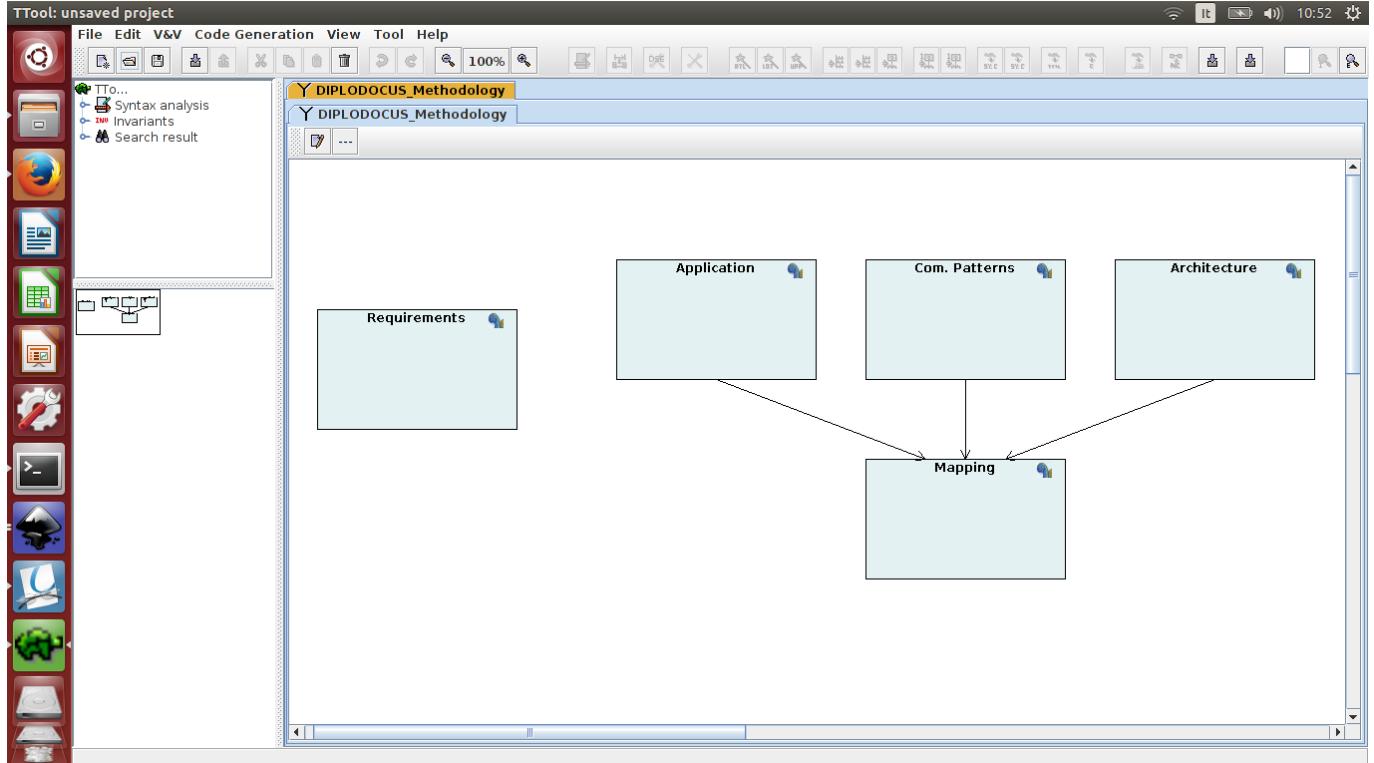
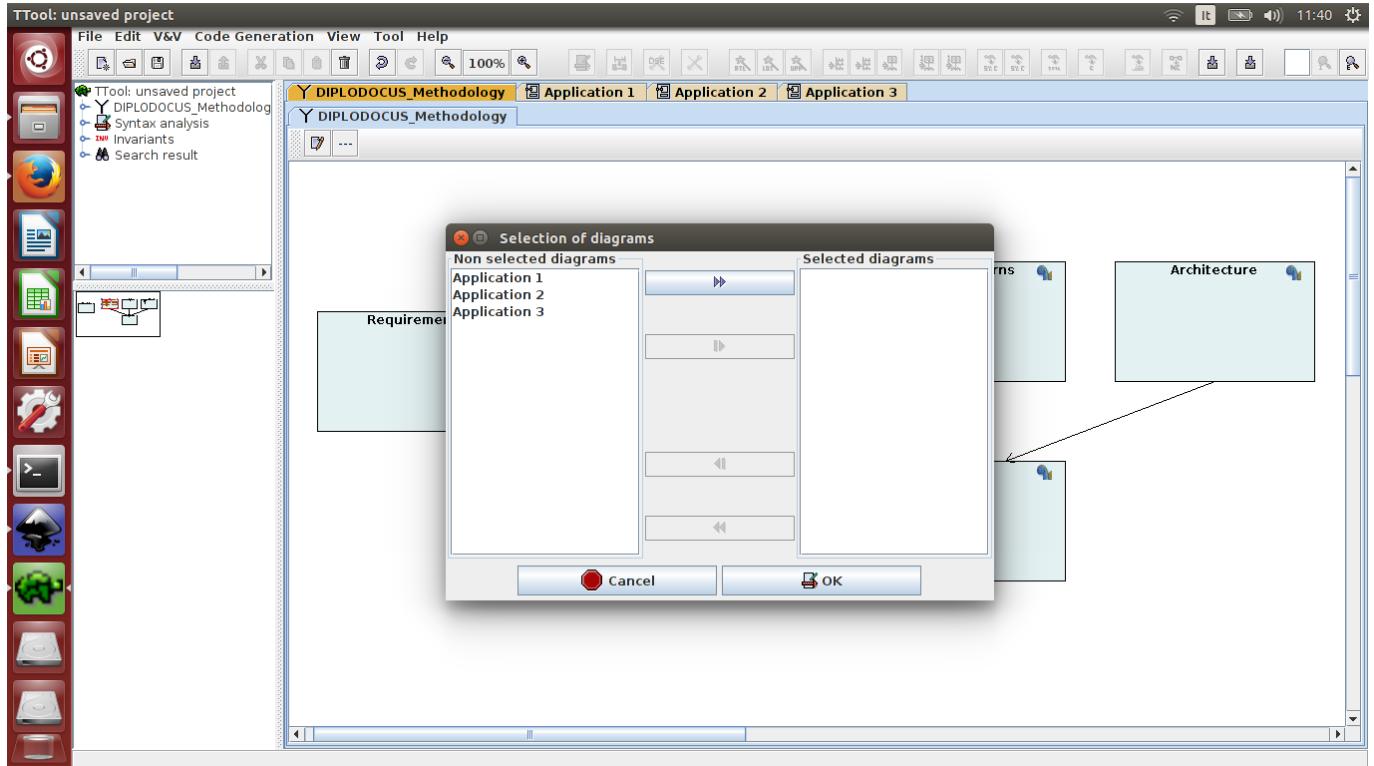
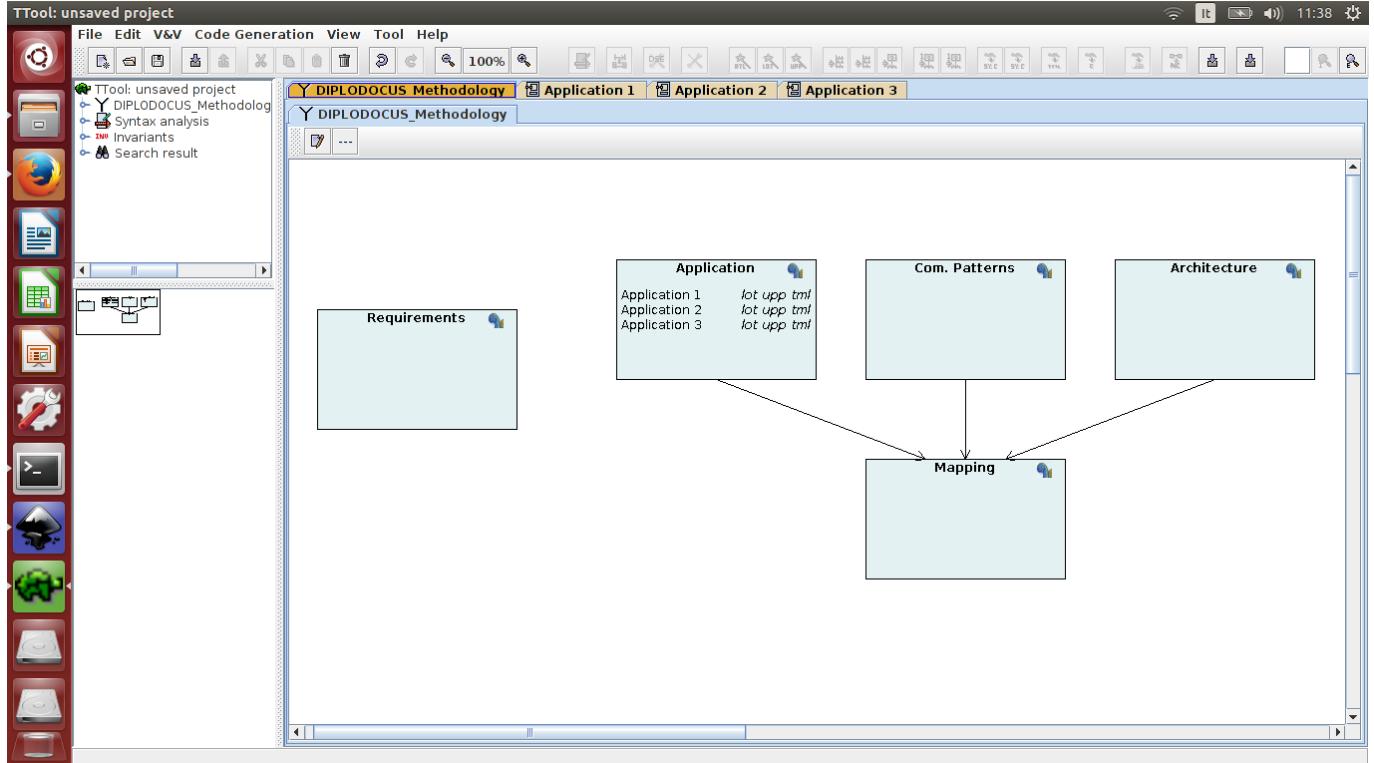


Fig. 6. The Methodology diagram of the  $\Psi$ -chart approach



**Fig. 7.** The graphical window to add a reference to a diagram to the Methodology diagram



**Fig. 8.** The Methodology diagram after the introduction of 3 references to 3 different application diagrams

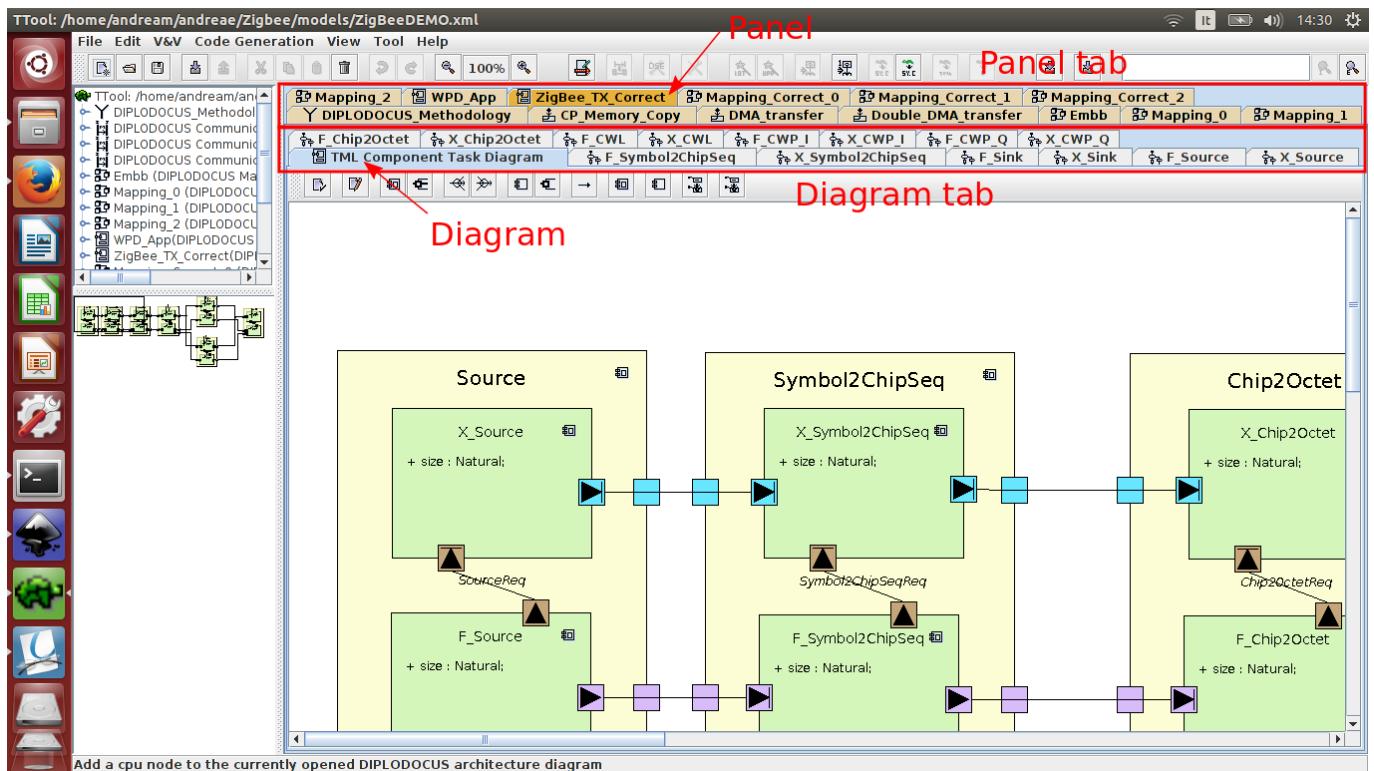


Fig. 9. The difference between a panel and a diagram

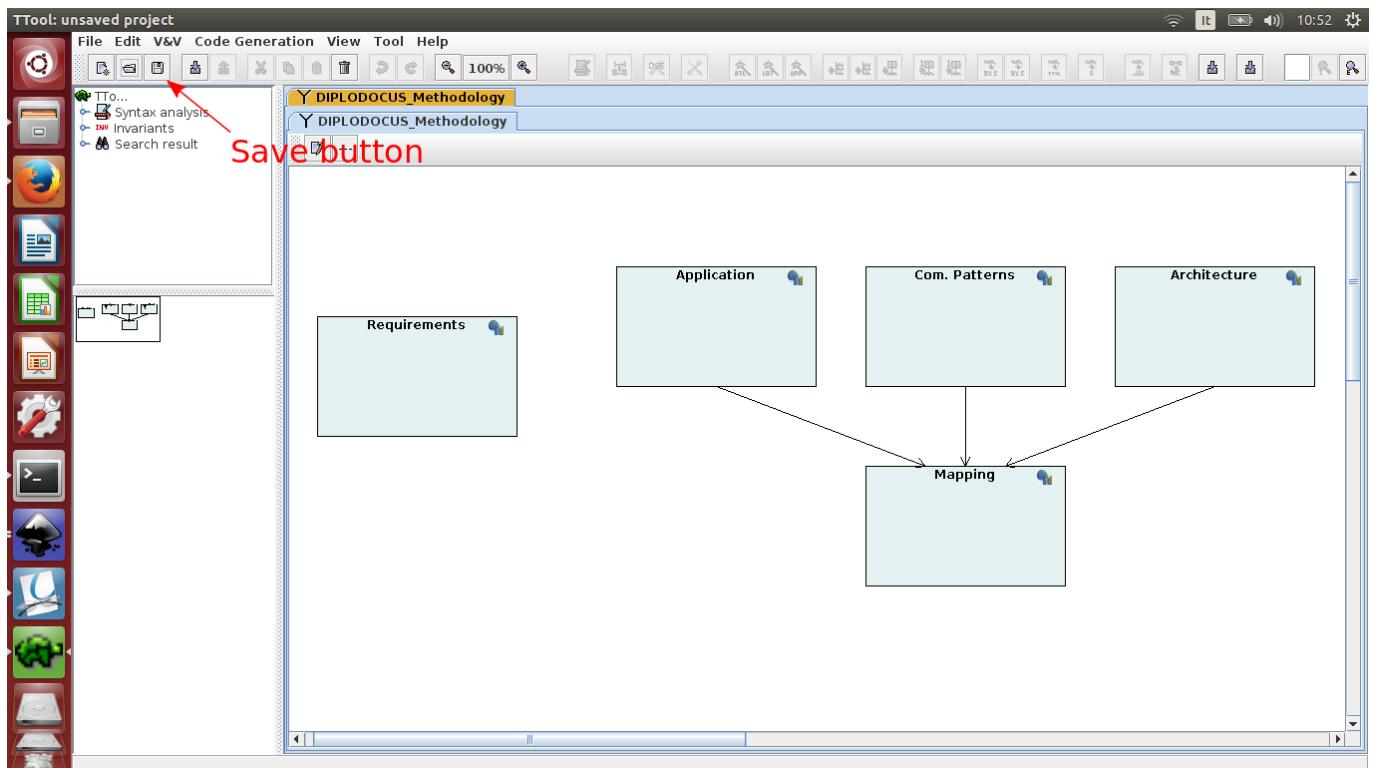
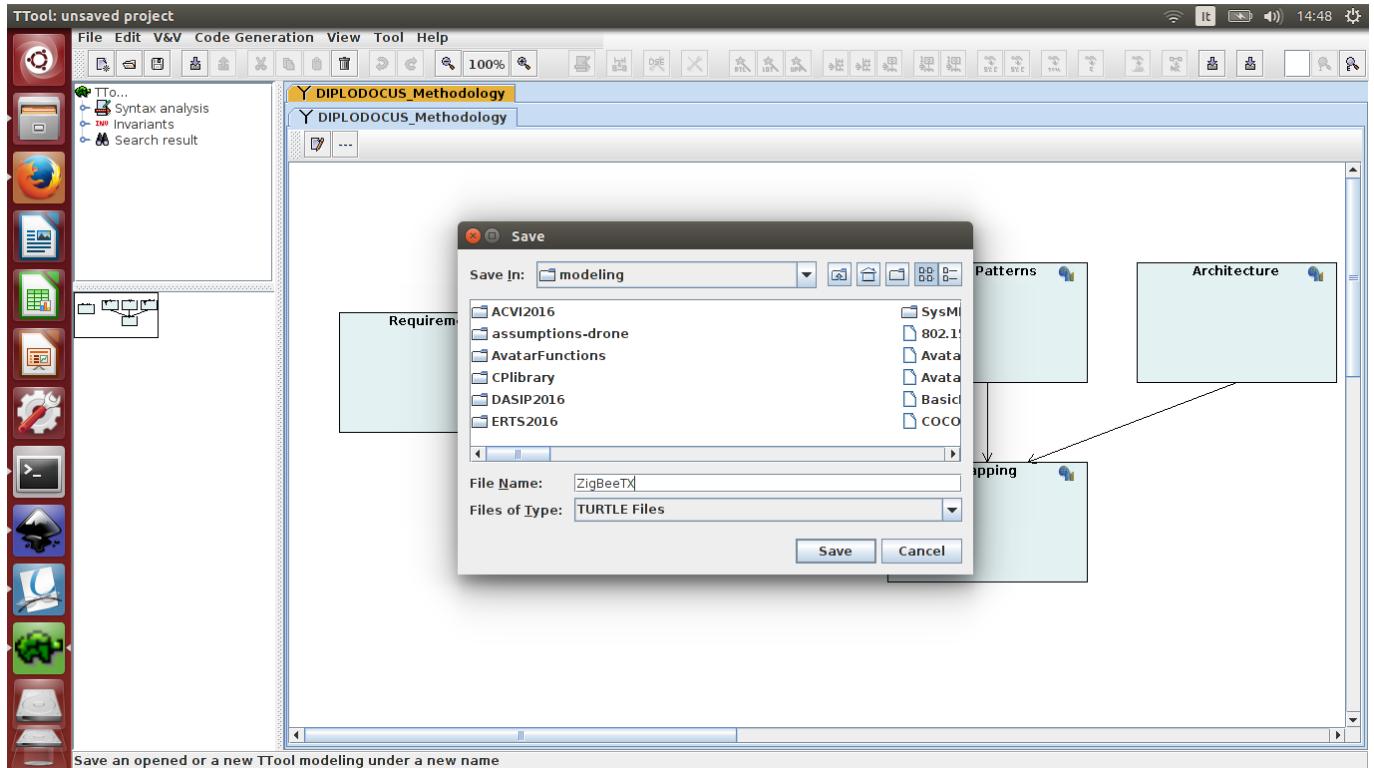


Fig. 10. The location of the Save button in the button bar



**Fig. 11.** Saving the project of the ZigBee transmitter

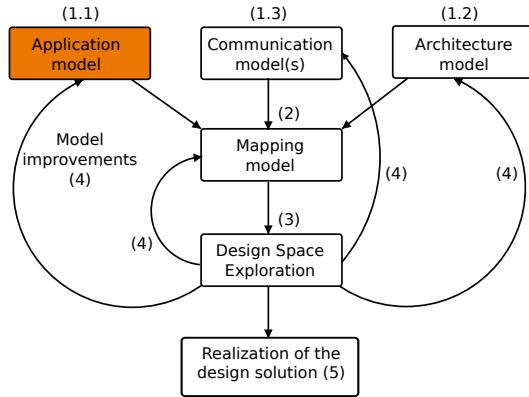
of the next sections is organized as follows: we will first introduce the reader to the theoretical aspects of the design (e.g., what Zigbee is, the architecture of the platform that we target). Subsequently, we show the complete diagrams that correspond to each part of the design or the tool windows that are used to perform a specific task (e.g., add a simulation breakpoint). We then illustrate to the reader how these diagrams can be created or these windows can be used.

## 7 Modeling a ZigBee transmitter

In this section we present the application, communication, platform and mapping models that we created specifically for the design of the ZigBee transmitter. We use this case study as an example to demonstrate the modeling facilities offered by TTool/DIPLODOCUS<sup>3</sup>

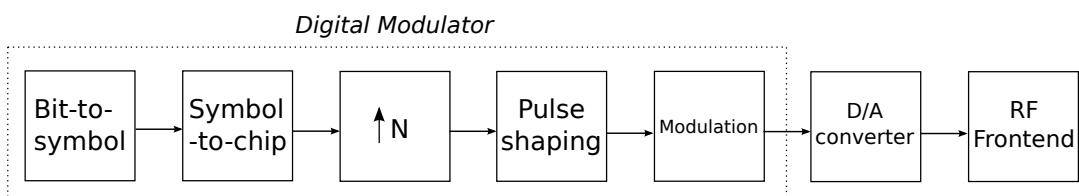
### 7.1 The functionality of a ZigBee transmitter (data-link layer)

Generally speaking, an application model (Fig. 12) captures the system's functionality (e.g., a signal-processing or video-compression algorithm). This model must be created regardless of the resources that are available for execution purposes (i.e., hardware and/or software resources) and must express all potential parallelism between operations. This model must express both the processing of information (e.g., computations) and the dependencies (e.g., communications) between these processing operations.



**Fig. 12.** The step of application modeling, that is described in this section, in the context of the  $\Psi$ -chart design approach

In this subsection we describe the application model for the data-link layer of a ZigBee transmitter. The ZigBee protocol is issued by the IEEE 802.15.4 standard that specifies both the MAC and the PHY layers of the IEEE 802.15.4 protocol. It is a standard for low-rate Wireless Personal Area Networks (WPANs), which are used to convey information over relatively short distances. ZigBee has been deployed for several applications including Wireless Sensor Networks (WSN) for building automation, remote control, health care, smart energy, telecommunication services. Among the different schemes that can be derived from the IEEE 802.15.4 standard for a ZigBee transmitter, we selected the one proposed by [11], shown in Fig. 13, because of its simplicity in terms of implementation.



**Fig. 13.** The functional block diagram of the ZigBee transmitter as proposed by [11].

Fig. 14 shows the TTool/DIPLODOCUS diagram corresponding to the data-flow model of Fig. 13 implemented for

<sup>3</sup> Note that the complete model of this tutorial is released with TTool and can be opened by clicking menu **File>>Open** and then selecting file **ZigBeeTutorial.xml** under directory **DIPLODOCUS** from the dialog box that opens.

EMBB. This means that the diagram of Fig. 14 expresses the functionality of the algorithm in Fig. 13, according to the signal-processing operations available in the platform EMBB (see sub-section 7.3). This statement may apparently contradict the above claim that a system's functionality and its resources are modeled separately. However, the reader must keep in mind that the functionality of a system (the ZigBee transmitter, in our case) can, and must, be modeled independently of the specific resources of a target platform (the processors, buses and memories of EMBB, in our case). On the contrary, it cannot be modeled independently of the services offered by these resources (the signal-processing operations offered by these processors, in our case). Frequently, these services, do not correspond to the mathematical operations of signal-processing algorithms, such as the one in Fig. 13.

In Fig. 14, the block labeled Source produces the data to be transmitted in the form of a flow of bits. These data are then converted to symbols by the Symbol2ChipSeq block. In this block, we model the mapping of each incoming 4-bits symbol to one of the 16 sequences of 32 chips as defined by the IEEE standard 802.15.4. The Chip\_to\_Octet block then transforms each incoming chip (bit) of a chip sequence into an unsigned 8-bits integer as expressed in equation 1:

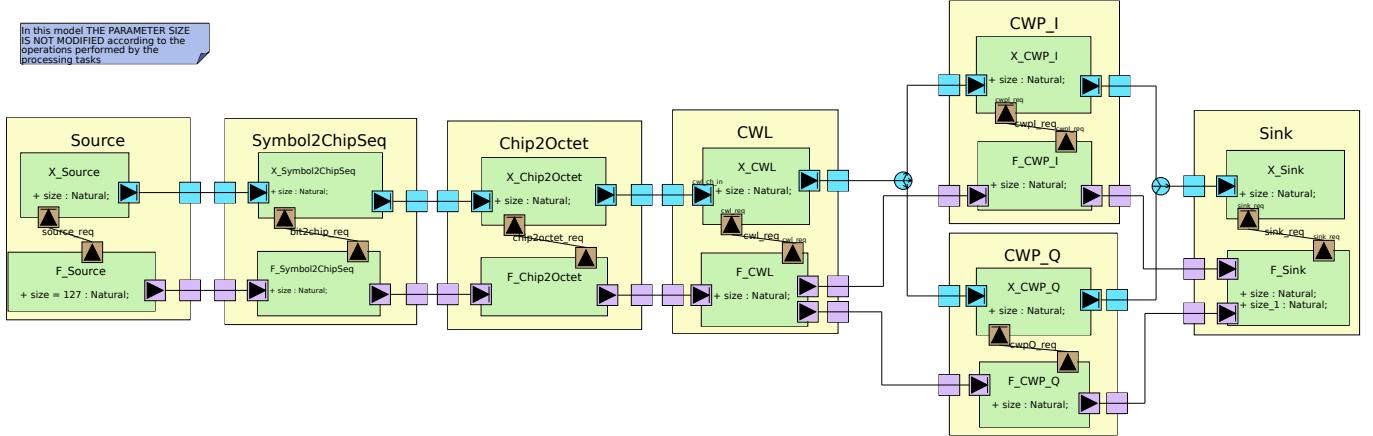
$$\{0; 1\} \rightarrow \{0x00; 0x01\} \quad (1)$$

Chip\_to\_Octet also separates the even-indexed chips that are used to modulate the in-phase (I branch) carrier component from the odd-indexed chips that are used to modulate the quadrature (Q branch) carrier component. The output is then transformed by means of a Component Wise Lookup (CWL block) that maps unsigned 8-bits integers to signed 16 bits integers as expressed by equation 2:

$$\{0x00; 0x01\} \rightarrow \{0xffff; 0x0001\} \quad (2)$$

At this point, given the separation of the I and Q branches, their pulse shaping can be executed independently. The application graph exposes this parallelism by forking the output data of block CWL to two distinct Component Wise Product (CWP) blocks, CWP\_I for the I branch and CWP\_Q for the Q branch. These blocks multiply the input samples with a half-sine wave to realize the O-QPSK modulation. The quadrature shift between the I and Q branches is implemented by means of an offset between the memory addresses of the output samples. This results into a frame of complex samples (16 bits for the real part and 16 bits for the imaginary part) that is then collected by block Sink and transmitted over the air.

Each block of the model in Fig. 14 is composed of two sub-blocks (tasks): one modeling the data-processing and one modeling the related control operations. By convention we name the data-processing tasks with a heading X that stands for eXecution and the control tasks with a heading F that stands for Firing.

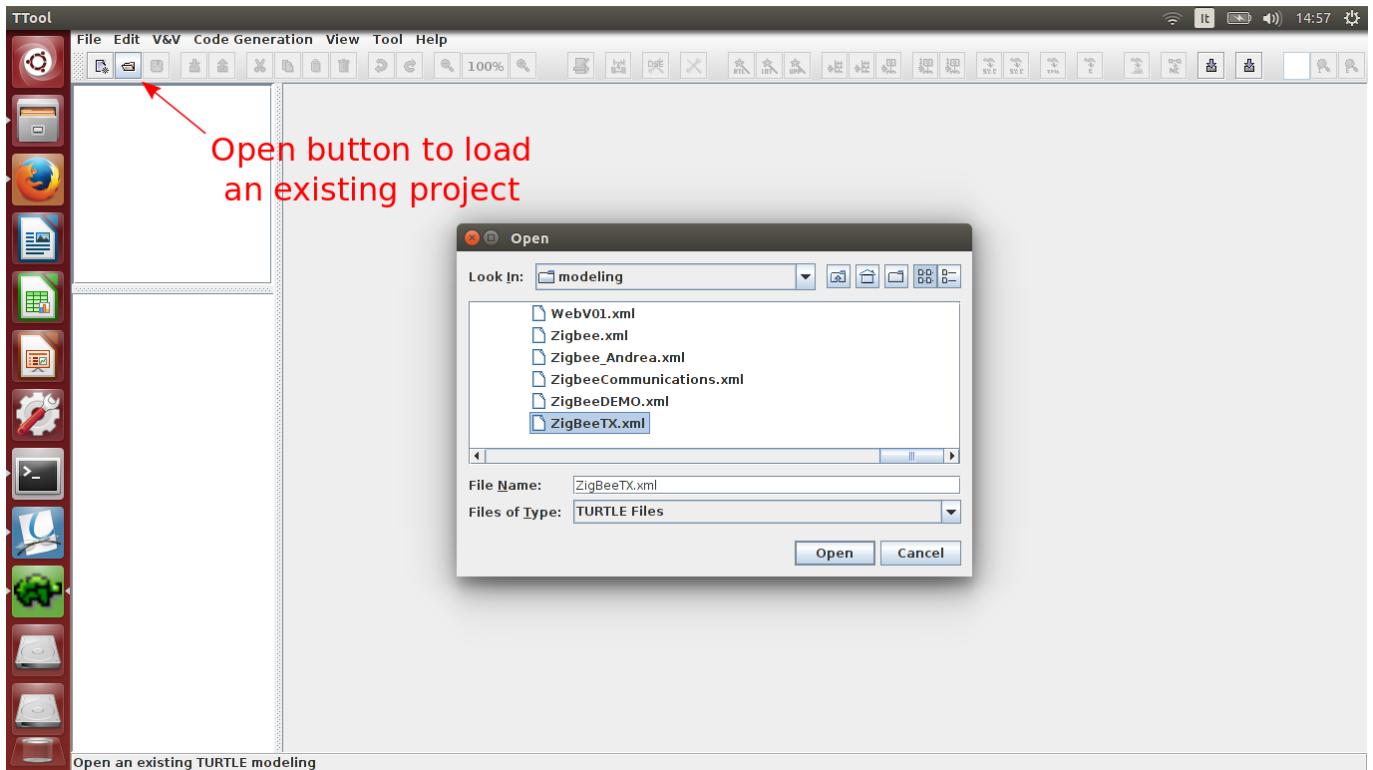


**Fig. 14.** The TTool/DIPLODOCUS model of the ZigBee transmitter

## 7.2 Creating the application model of a ZigBee transmitter (data-link layer)

Let's now reload the ZigBee project that we have created in Section 6. This can be done by clicking on the dedicated button, Fig. 15, or by selecting **File->Open**.

To create the application model in Fig. 14, we first need to create an application panel and then start adding the related



**Fig. 15.** The location of the **Open** button in the button bar

diagrams. For this purpose, let's right click in the panel tab and select **New Partitioning - Functional view**. The tool will create a panel that contains a sub-panel where the user can draw the SysML Block Definition diagram that captures the structure of the system's functionality, Fig. 16, in terms of interconnected blocks. The default name of this sub-panel is **TML Component Task Diagram**. As introduced in Section 2, TML stands for Task Modeling Language. It is a textual language, alternative to the graphical description provided by UML/SysML diagrams. In this version of the tutorial, we do not cover the syntax and semantics of the language. Some of the TML syntax and semantics is covered in [26] and in [18].

To rename the application panel (we suggest that you rename it **ZigBeeApp**), right-click on the panel and select **Rename**. Let's now start to build the application model of the data-link layer ZigBee transmitter! There are many approaches that can be taken to express the mathematical algorithm in Fig. 13 in a set of connected data-processing and control operations. The approach that we used, resulting in the diagram in Fig. 14, is to separate the data-processing operations and the control operations in independent blocks (*primitive components* in the language of DIPLODOCUS). Another approach would be to use only a single primitive component to represent both data-processing and control aspects. However, this modeling approach would result in several issues at the mapping step. In fact, the architecture of EMBB separates the control part from the data-processing in physically distinct units. Instead, for each of the signal-processing instructions that are available in EMBB, we instantiate a *composite component* that contains two primitive components. The latter are named with the prefix **X\_** (eXecution) for the data-processing part and with the prefix **F\_** (Firing) for the control part. In the frame of this modeling approach, execution of the **X\_** component is triggered by the **F\_** component.

An application model in TTool/DIPLODOCUS must always include a *Source* and a *Sink* components. The *Source* component emits the data to process and the related control information. The *Sink* component, instead, collects both data and control items. To draw the *Source* component as in Fig. 14, left-click on the **Add a composite component** button in the **TML Composite Task Diagram** panel. Then left-click in the design area where you want the composite component to be placed, Fig. 17. Double-click on the component and re-name it as *Source*. Enlarge the composite component, then instantiate two primitive components, one for the data part (eXecution) and one for the control part (Firing), by left-clicking on the **Add a primitive component** button. Place the two primitive components within the composite component. The primitive components are automatically attached to the composite component (i.e., they cannot be moved outside of the composite component). Rename the primitive components as **X\_source** and **F\_source** by double-clicking on the the component's default name, then type the new name. Please notice that two new tabs

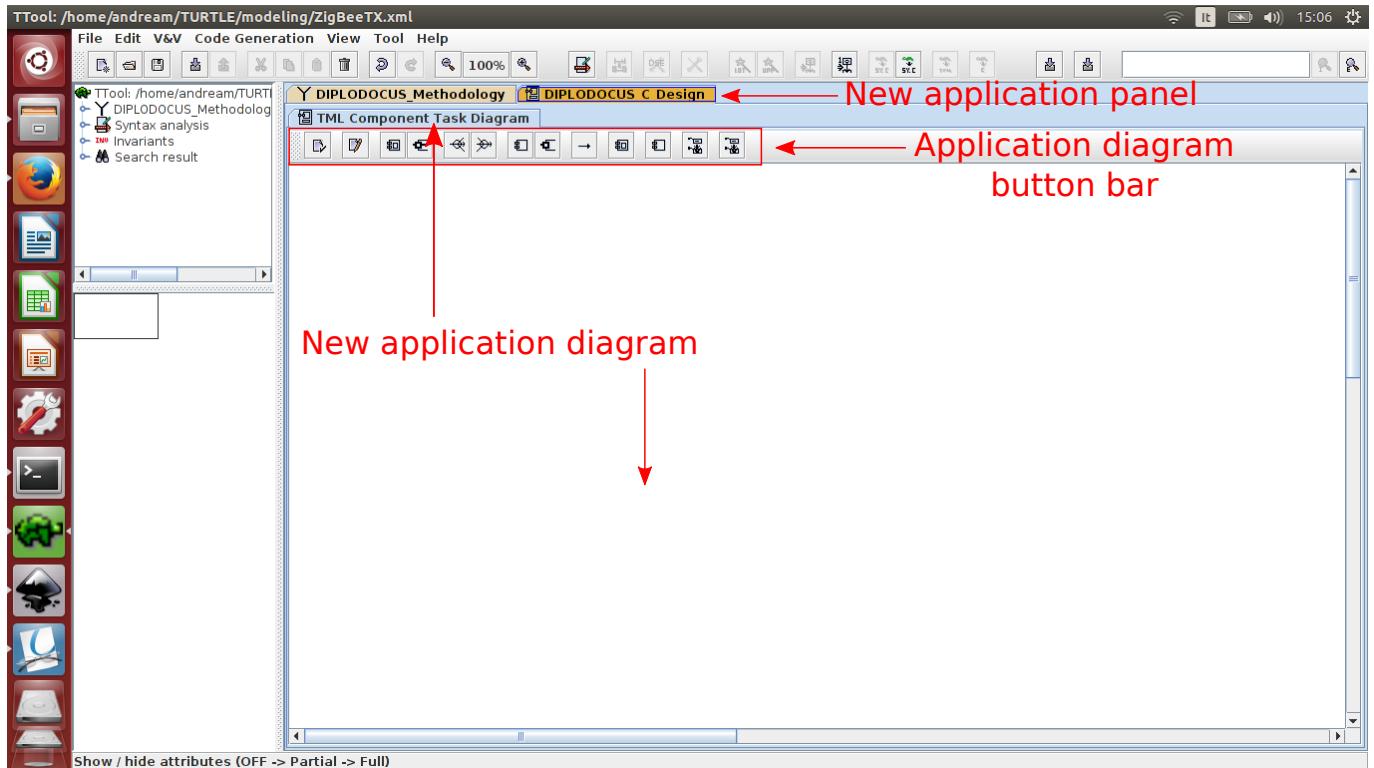


Fig. 16. The creation of a new application panel and diagram

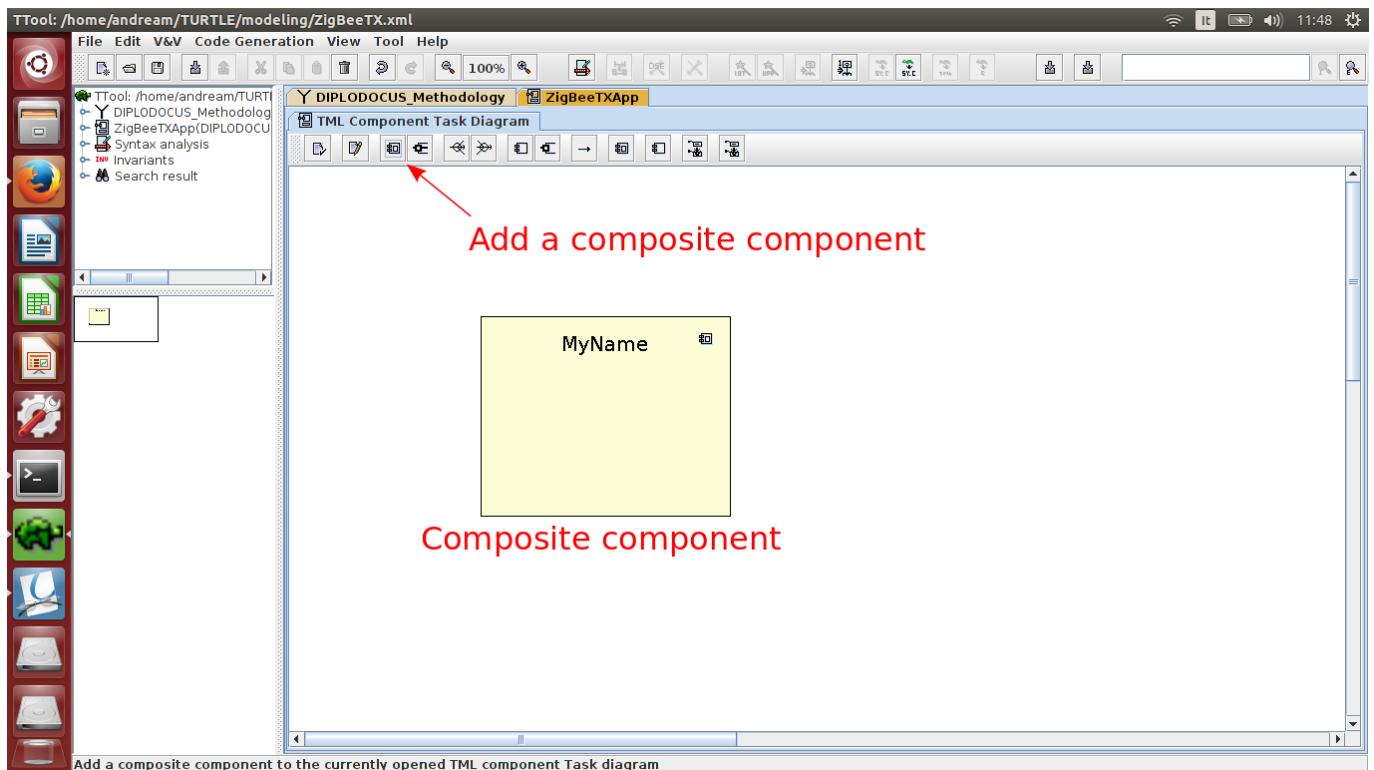


Fig. 17. The instantiation of the Source composite component

appear on the right-hand side of tab TML Component Task Diagram. We will use these tabs later on in this tutorial to design the activity diagram internal to each primitive component.

**Attributes of a primitive component** While composite components are simply containers for primitive components, the latter have attributes and contain a SysML Activity Diagram. One parameter that must be specified for this design is the number of data samples that our model will process (**size**). Control variables or attributes such as **size** are used by the control-flow operators of an activity diagram. Therefore, for each F-primitive component that uses **size**, we must declare such a control variable. This is done by double-clicking in the green area of a primitive component, as shown in Fig. 18. We define the parameter **size** as of type natural and assign it the default value of

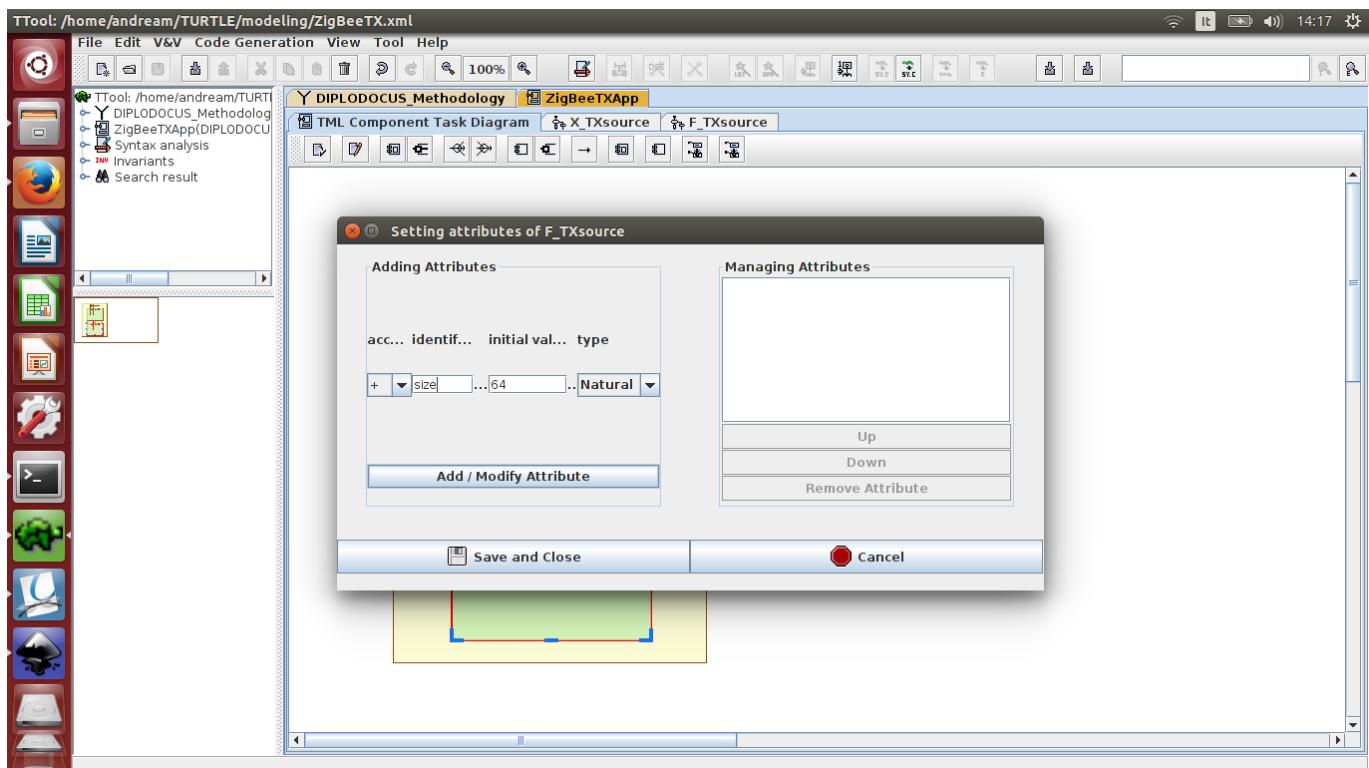


Fig. 18. The creation of the control parameter **size**

31 as in this case study we analyze the transmission of a ZigBee packet that is composed of 25 bytes payload and 6 bytes header. Then, click on the ADD/Modify attribute button. Clicking on the Save and Close button will add the variable to the primitive component that will list it in its green area. The parameter **size** must be created for both X\_source and F\_source primitive components, but for the X\_source component, it is not necessary to specify a default value (to define the control attribute) as it will be transmitted by F\_source. For component X\_source, we only need to declare **size**. The scope of an attribute is local to the diagram of a primitive component. So far, it is not possible to define or declare global attributes. The value of an attribute can be modified in a primitive component's diagram. Additionally, the value of an attribute can be exchanged among primitive components as described in the next paragraph.

**Ports, channels, events and requests** In TTool/DIPLODOCUS components communicate via channels, events and requests. More in detail, channels are used to exchange data, whereas events and requests are used to exchange control information. Events are used to synchronize control-flows, whereas requests are used to spawn the execution of primitive components. These three types of communications are attached to components via ports. Two types of

ports exist: composite ports for composite components and primitive ports for primitive components.

Channels are characterized by a point to point communication between 2 tasks<sup>4</sup>. The possible types for a channel are:

- Blocking Read - Non Blocking Write (BR-NBW): the emitter task can write infinite times while the receiver task blocks when attempting to read from an empty channel. Data are read in a FIFO. Therefore a BR-NBW channel is equivalent to an infinite FIFO buffer.
- Non Blocking Read - Non Blocking Write (NBR-NBW): the emitter task can write infinite times and the receiver task never blocks when attempting to read from an empty channel. A NBR-NBW channel is equivalent to a shared memory of infinite size between emitter and receiver.
- Blocking Read - Blocking Write (BR-BW): the emitter task blocks when attempting to write to a full channel and the receiver task blocks when attempting to read from an empty one. A BR-BW channel therefore is equivalent to a finite FIFO buffer.

When configuring a port to be for a data channel, the user must specify the number of data samples (to be written or read), and the maximum number of samples to be written before blocking (this second one for BR-BW only).

Events are characterized by a point to point asynchronous unidirectional communication between two tasks. Multiple events arriving at the same task at a given moment are managed using a FIFO, which can be finite or infinite.

- In the case of infinite FIFO, events are never lost.
- When using a finite FIFO, events arriving at the FIFO are stored in it. Two semantics are defined for finite FIFO:
  - When the FIFO is full, the first (oldest) element is removed to leave space for the new one that is added.
  - When the FIFO is full, no event may be added: the event sender is blocked until the FIFO is not full.
- In the case of a single element FIFO, with first event removal, it is equivalent to a hardware interrupt or a Unix signal.

Up to five optional parameters can be specified for an event, and they can be Integer or Boolean. The Boolean type is defined as an Integer, where the integer value '0' means false, while any other integer value means true.

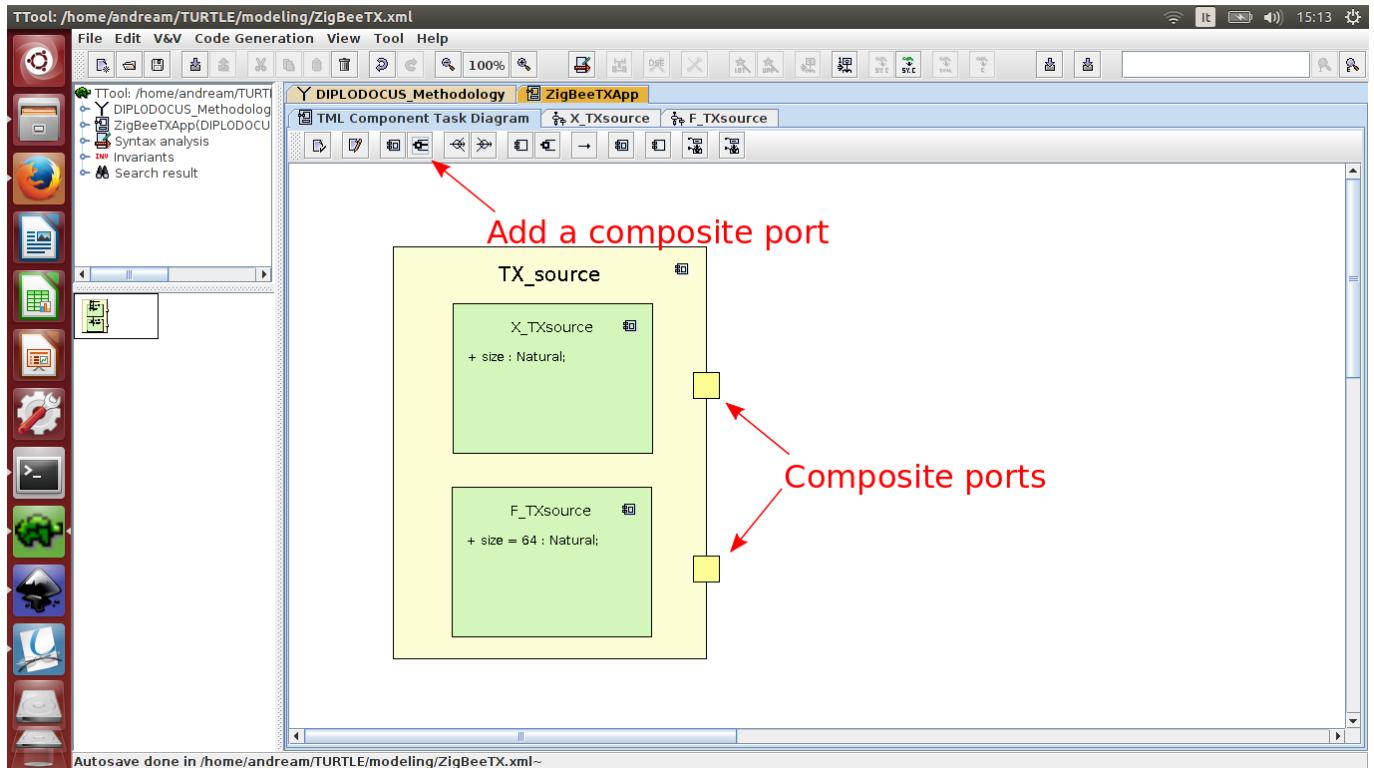
Requests are characterized by a multipoint to one point asynchronous unidirectional communication between two tasks. Multiple requests arriving to one task at a given moment are stored in an infinite FIFO (a FIFO is defined for each destination task). They can be executed straight or after the previously stored requests have been executed. In any case, since the FIFO is infinite, requests are never lost. Requests are never blocking for the sender task. Up to five optional parameters can be specified for a request, and they can be Integer or Boolean.

As the component in Fig. 18 is the source of our design, we only need to add output ports. In this case, the two primitive components have each one output flow: one to send data and one to send control information. Therefore, we need to instantiate 2 primitive ports and 2 composite ports as in Fig. 19 and Fig. 20. To add a composite port to your design, left-click on the **Add a composite port** button and then place the port anywhere within the boundaries of the composite component, it will be automatically attached to its edges. Then drag the port on the right-hand side of the component as in Fig. 19. To add a primitive port, left-click on the **Add a primitive port** button and then place the port anywhere within the boundaries of the primitive component, it will be automatically attached to its edges. Then drag the port on the right-hand side of the component as in Fig. 20. By default a port is instantiated as a channel port (blue port) and is configured as an output port (rightward black arrow-head). The tool displays the name associated to the port, that by default is **comm**. To link a primitive port to a composite port, left-click on the **Add a connector between two ports** button. This will automatically highlight connection points in the design (yellow squares on ports). Left-click on the connection point of a primitive port and then left click on the connection point of a composite port. Please notice that while primitive ports only have one connection point, composite ports have both an input (leftmost) and an output (rightmost) connection point. Therefore, pay attention to connect the primitive ports to the input connection point of a composite port. At this point in our design, the composite ports are not completely connected so the tool will color them in red to highlight this syntax error. The latter can be safely ignored for the moment. The primitive ports are also partially colored in red, Fig. 22. For the **X\_** primitive component, it is correct to have a data port, but for the **F\_** port we must change the port type to that of an event. This way, the component will be able to exchange attribute **size** with other components. Double click on the port of component **F\_source**, rename it as **SourceEvtOut**, select the type to be an **Event** and the first parameter (Type #1) to be of type **Natural**, Fig. 21. Then save and close the window. For the port attached to component **X\_source**, double click on it and rename it as **SourceChOut**. Then select the port to be of type **Blocking**.

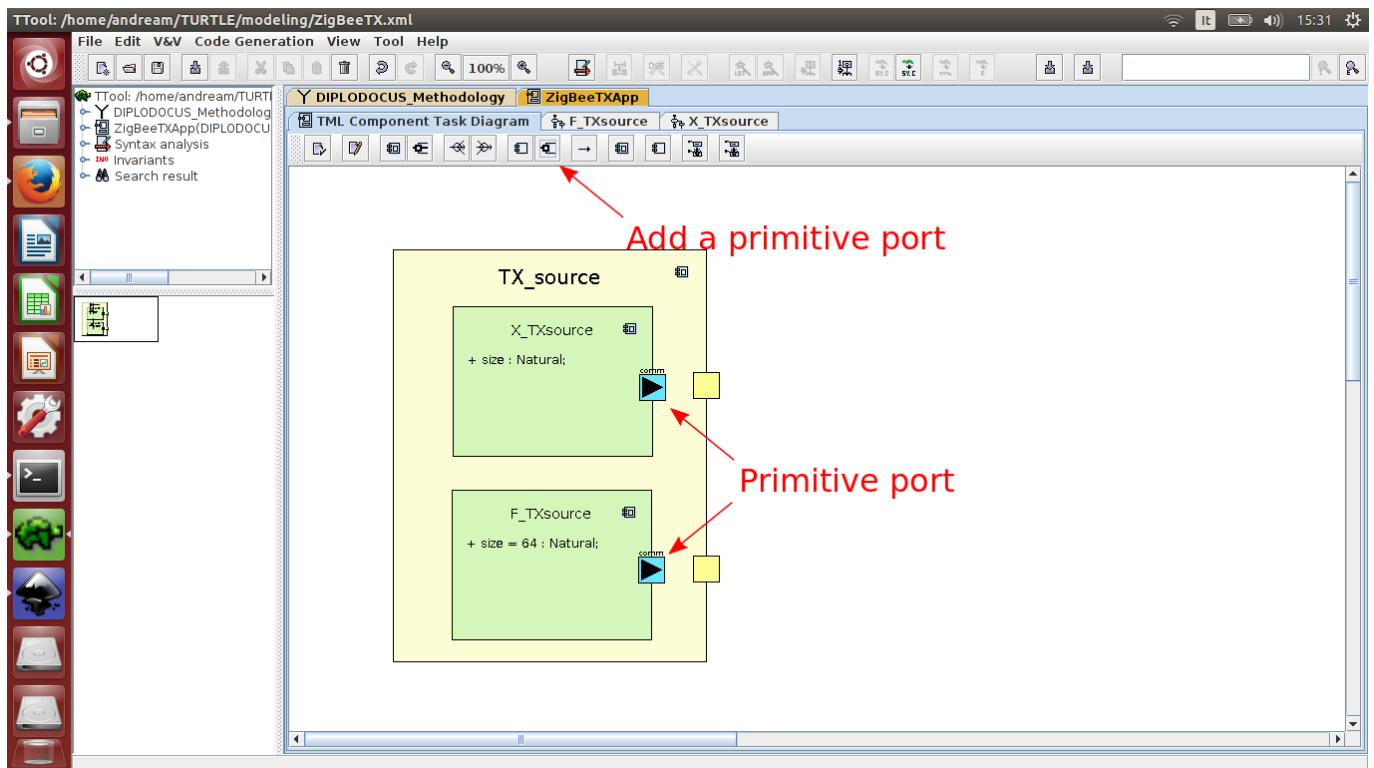
As mentioned above, it is the Firing primitive component that triggers the eXecution primitive component. Therefore,

---

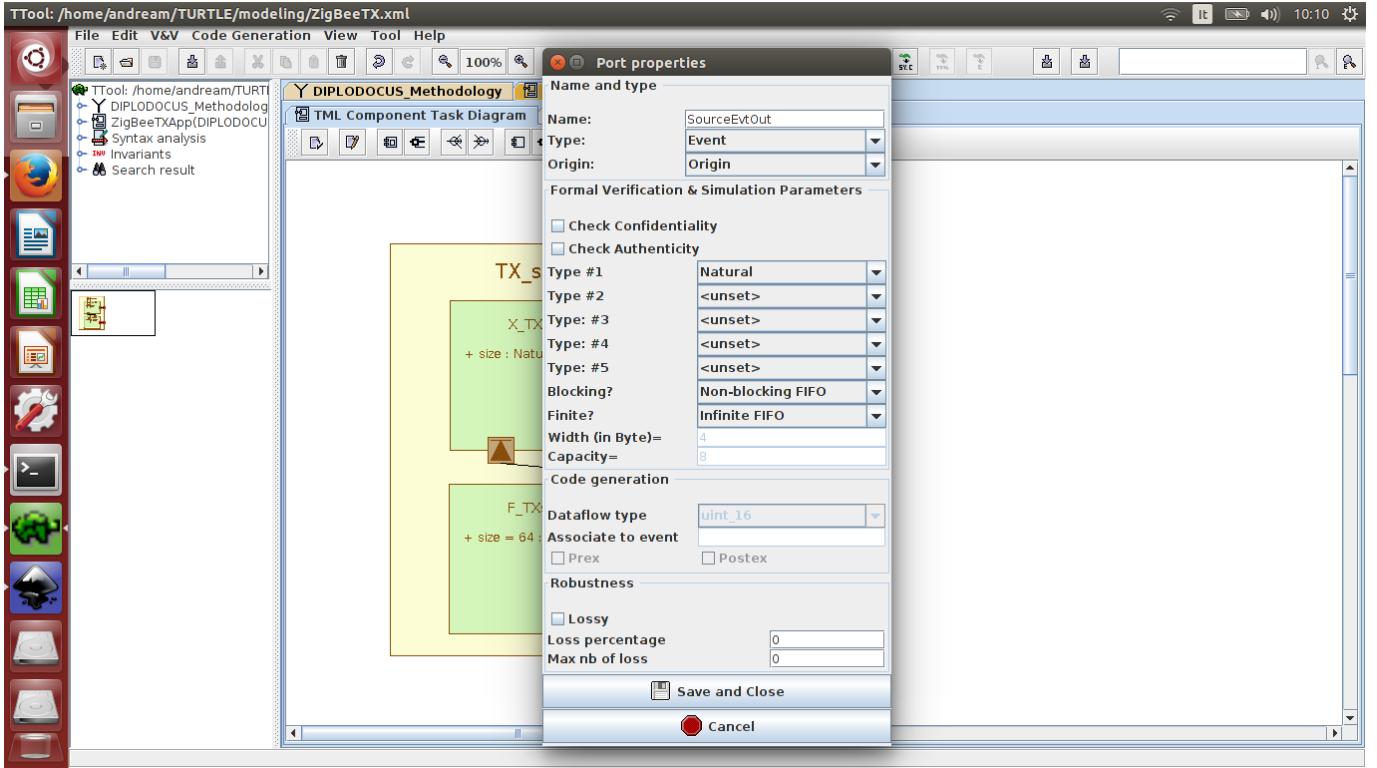
<sup>4</sup> A task is a primitive component in the terminology of the TML language



**Fig. 19.** The instantiation of composite ports to the composite component **Source**



**Fig. 20.** The instantiation and configuration of ports for the primitive components

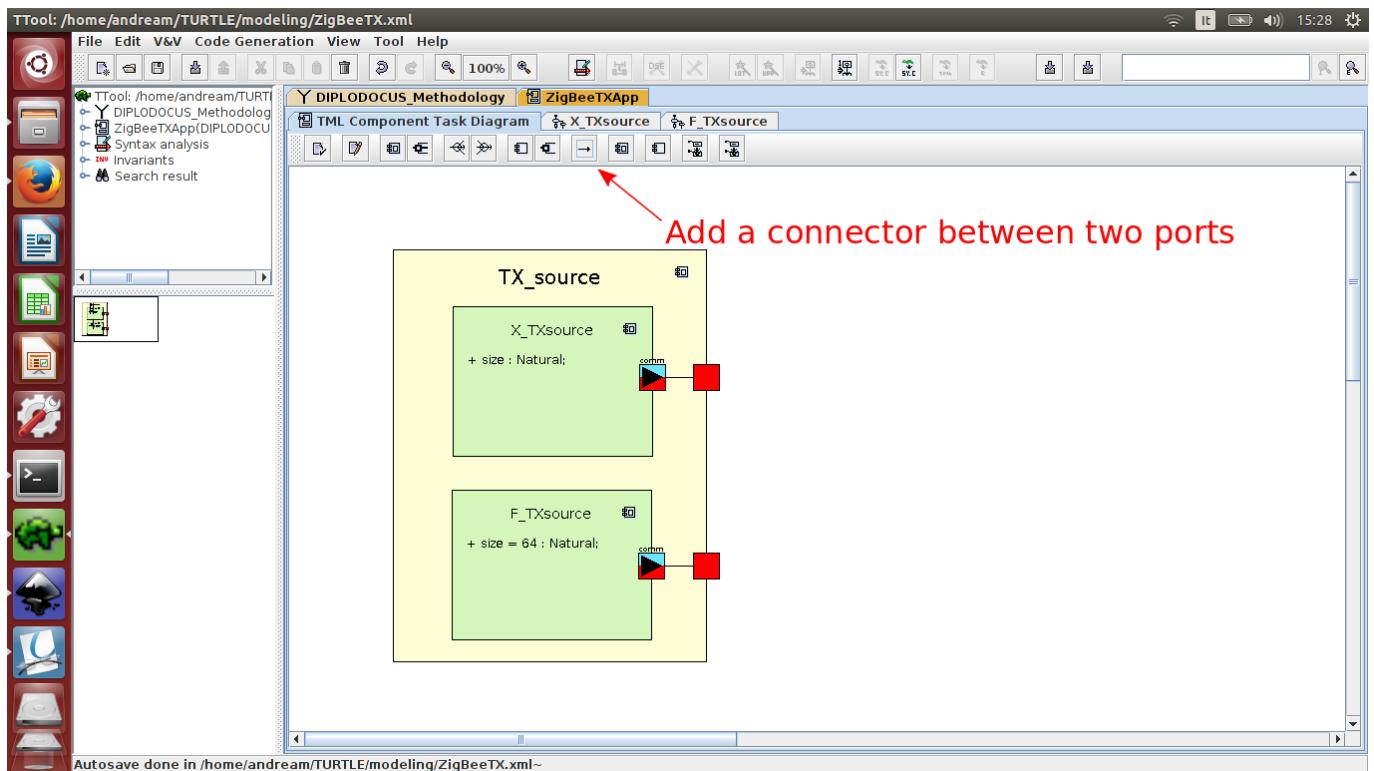


**Fig. 21.** Configuring the output event port of `F_source`

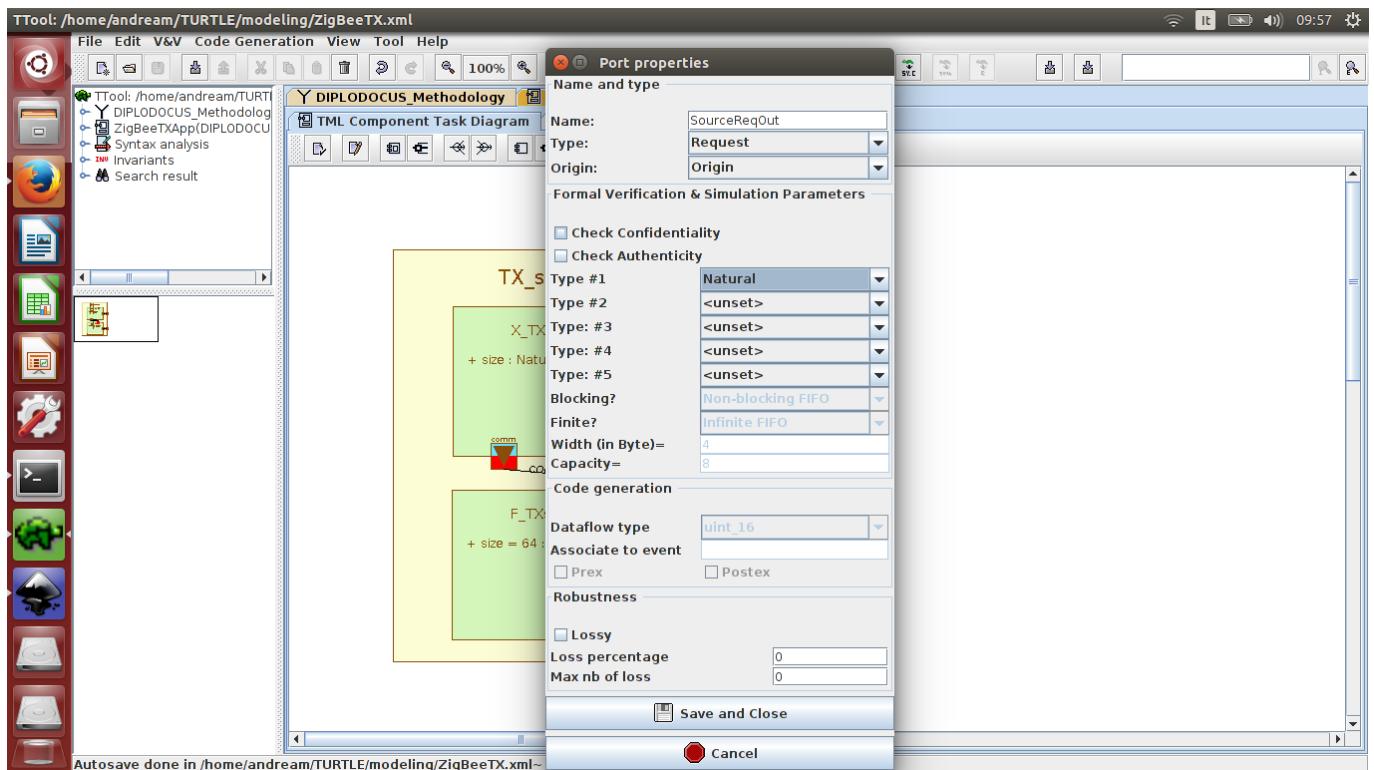
the two components must be interconnected by a request that `F_source` dispatches to `X_source`. To implement this in TTool/DIPLODOCUS, add another primitive port to each of the primitive components and connect them. Then double click on the port attached to component `F_source`, Fig. 23. Rename it as `SourceReqOut`, select the type to be a `Request` and the first parameter (Type #1) of type `Natural`, then save and close the window. Similary, rename the port attached to `X_source` as `SourceReqIn`, select the type to be a `Request`, select the origin to be a `Destination` port and the first parameter (Type #1) of type `Natural`, then save and close the window. Requests can carry up to five parameters; in our case this request is used to transmit the parameter `size` of type `natural`. The remaining buttons used to draw a `TML Component Task Diagram` (functional view of a design) are enumerated in Fig. 24 and described below:

1. Edit TML Task Activity Diagram
2. Add a comment: add a comment to the diagram
3. Add a channel fork: allows an input channel to fork in up to 3 output channels
4. Add a channel join: allows up to 3 input channels to join into 1 channel
5. Add a reference to a composite component: allows to reference a composite component from another application panel
6. Add a record component
7. Show/hide internal components: shows the internal components of an application diagram, e.g., parameters of events and requests
8. Show/hide DIPLODOCUS IDs: show or hide the identification (natural) numbers associated to each operator

**The activity diagram of a primitive component** As mentioned above, each primitive component has optional attributes and an activity diagram. The latter describes the internal functionality of the component (like a state machine): the processing of data, control items and the reaction to external events such as the reception of data or control information. In TTool/DIPLODOCUS, each primitive component must be associated to an activity diagram. To open the diagram window we can either click on the dedicated icon within the area of a primitive component, or open its corresponding panel as shown in Fig. 25. Let's first draw the diagram for component `F_source`. The design window for the diagram is displayed in Fig. 26. You can go back to the main diagram by right-clicking on the design



**Fig. 22.** Connecting primitive to composite ports



**Fig. 23.** Configuring the port of F\_source as a request

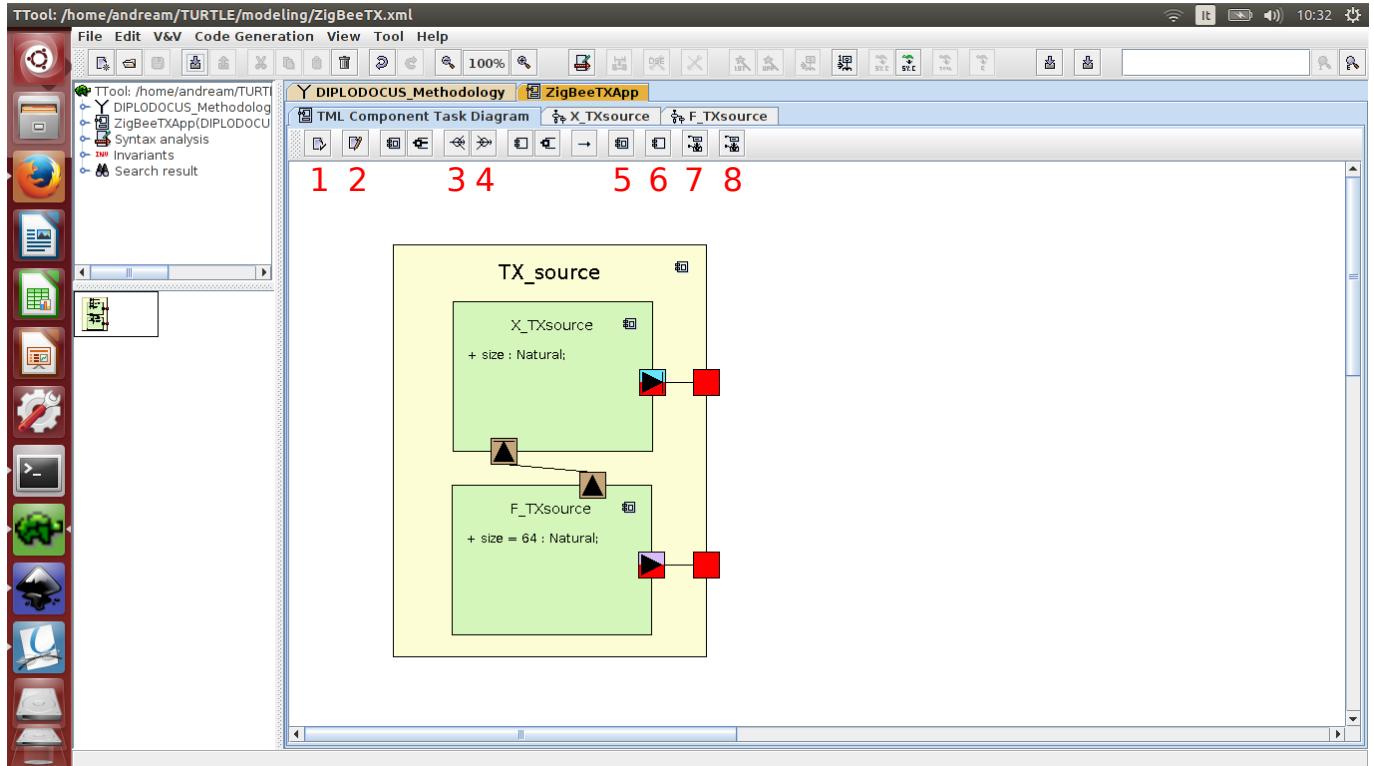


Fig. 24. The buttons for the TML Composite Task Diagram enumerated

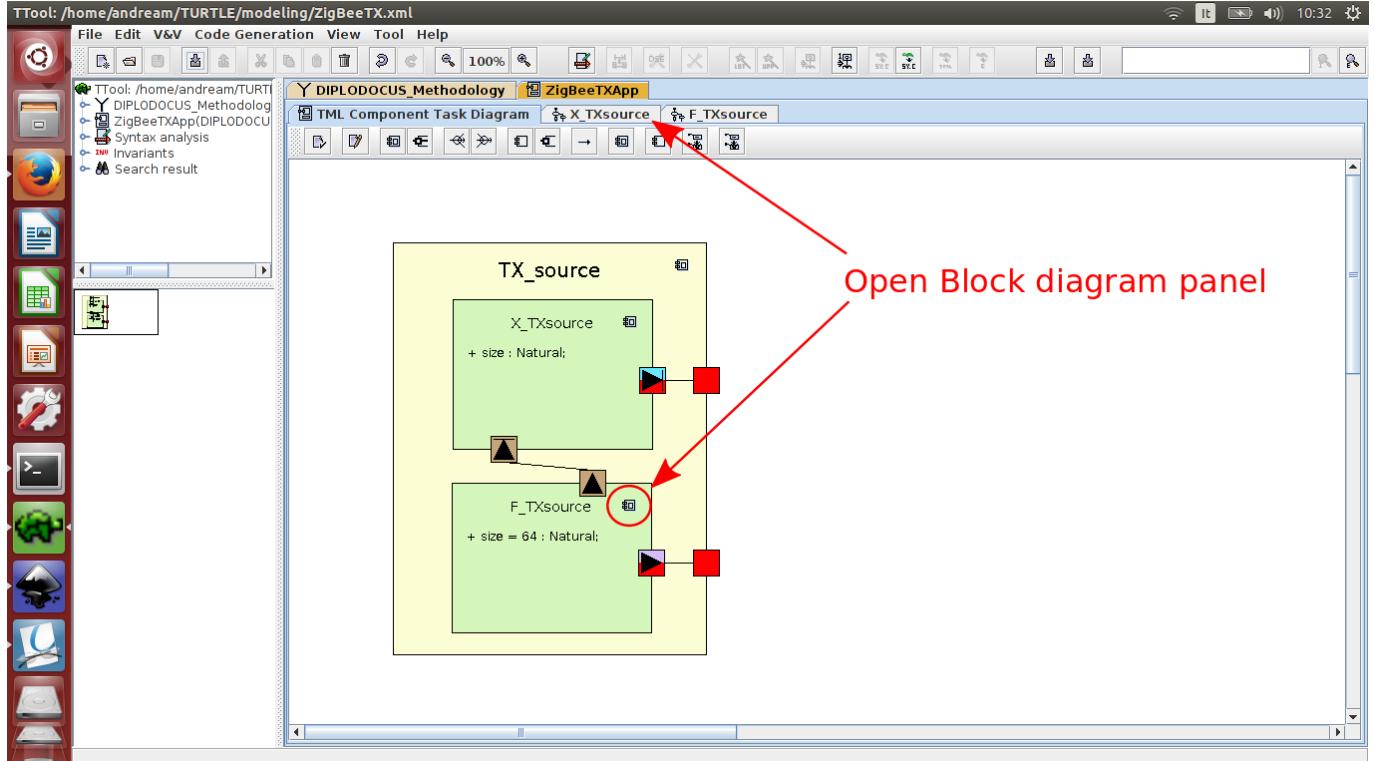
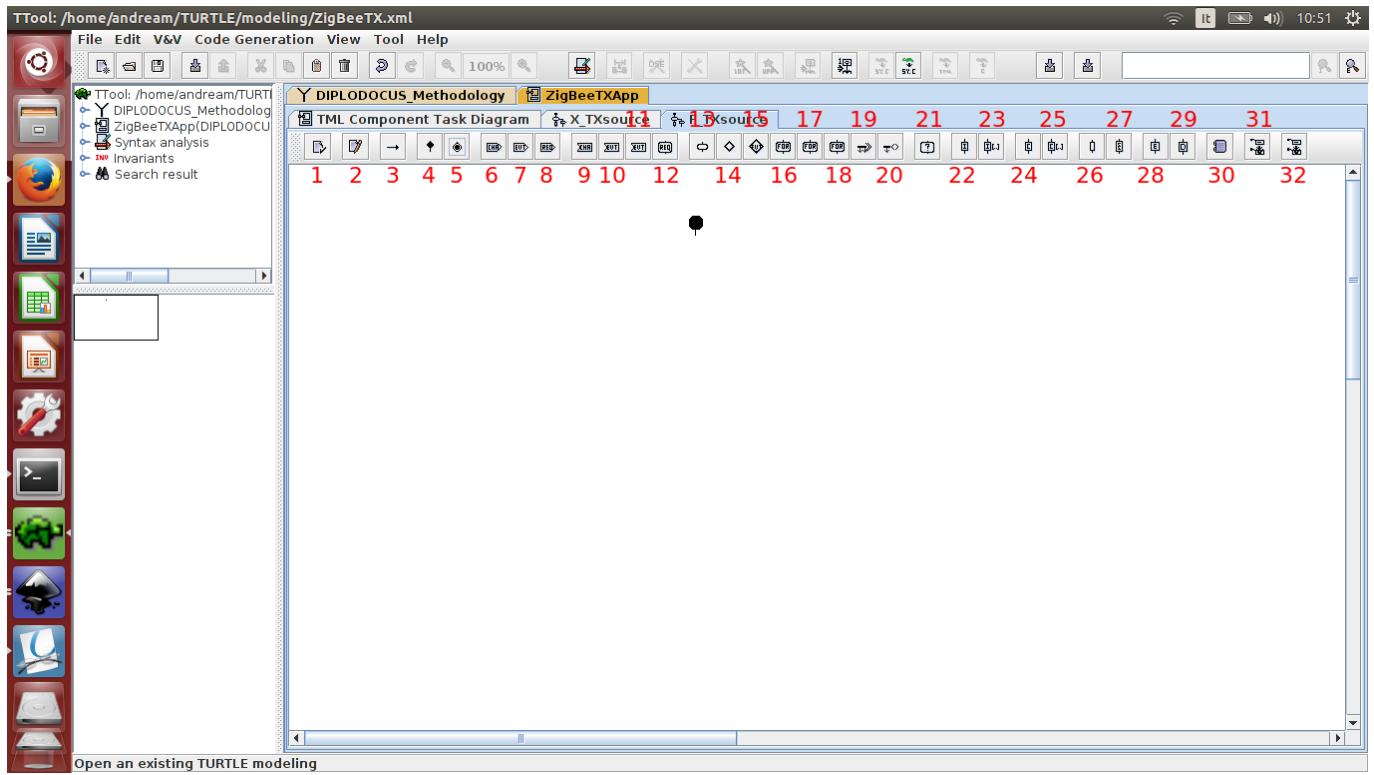


Fig. 25. Open the block diagram panel

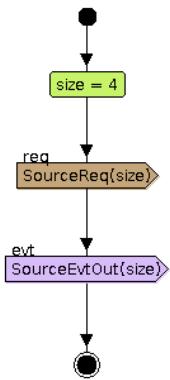
window and use “Back to main diagram” option. In Fig. 26, we have enumerated the buttons that are available to construct the diagram:

1. Edit TML Task Activity Diagram
2. Add a comment: add a comment to the diagram
3. Connect two operators together: link two operators
4. Start: the starting point of a diagram’s execution
5. Stop: stop the execution of the diagram or mark the end of a branch in a control structure (e.g., body of a for loop)
6. Write in channel: write a given amount of data to an output channel. Can be blocking or non-blocking according to the type of the event link (selected by user)
7. Send event: send an event and its associated attributes
8. Send request: send a request
9. Read in channel: read incoming data from a channel. Can be blocking or non-blocking according to the type of the event link (selected by user)
10. Wait event: wait for an incoming event. Can be blocking or non-blocking according to the type of the event link (selected by user)
11. Notified event: retrieve the number of incoming events received by a port. This operator tests if the event FIFO is not empty and returns a boolean value: if the FIFO is empty returns false, else returns the number of events in the FIFO.
12. Reading request arguments: retrieve arguments from a request and store the corresponding value in attributes
13. Action state: take an action on an attribute, e.g., increase/decrease its value
14. Choice: conditionally branch the execution of the diagram, according to boolean expressions
15. Select event: conditionally branch the execution of the diagram, according to the availability of events connected to the operator. This operator waits for one of the associated events to be available in the event FIFOs. In case several events are available in different FIFOs, then one among possible is randomly chose and read (consumed).
16. Loop (for): a classic for loop such as in the C language: `for(i=0;i<5;i++)`
17. Static loop (for): a loop contract that allows the user to select the number of iterations, e.g., `loop 10 times`
18. Loop for ever: loop indefinitely
19. Sequence: executes each interconnected outgoing branch in sequence.
20. Random sequence: randomly select an available operator
21. Select random: assign to an attribute a random value with uniform probability, given a range of values
22. EXECI: express the complexity of a data-processing operation as the number of computations on integer values
23. EXECI (time interval) express the (unknown) complexity of a integer data-processing operation as a range of computations on integer values
24. EXECC express the complexity of a data-processing operation as a range of computations on complex values
25. EXECC (time interval) express the (unknown) complexity of a complex data-processing operation as a range of computations on integer values
26. DELAY express the complexity of a data-processing operation in terms of time units
27. DELAY[,] express the (unknown) complexity of a data-processing operation in terms of a range of time units
28. Encryption: it can be used to indicate forging a Cryptographic Configuration and additional processing overhead due to the presence of security properties, see Section 10.
29. Decryption: it can be used to recover encrypted data and indicate additional processing overhead due to the presence of security properties, see Section 10.
30. Enhance
31. Show/hide internal components: has no effect on this diagram
32. Show/hide DIPLODOCUS IDs: show or hide the identification (natural) numbers associated to each operator

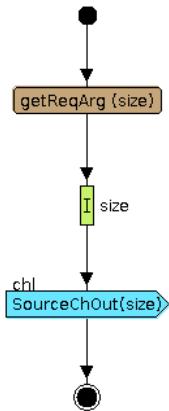
The activity diagram of component `F_source` is depicted in Fig. 27. The diagram in Fig. 27 is composed of the following operators interconnected in sequence: an action, the dispatch of a request, the dispatch of an event. To instantiate these operators, simply click on the corresponding button (Fig. 26) and then place the operator in the design area. To configure each operator, simply double click on it, then enter the desired parameters or select the desired ports for channels, events and requests. Do not forget to properly link the operators by means of the interconnect operator. The activity diagram of component `X_source` is depicted in Fig. 28. It is composed of the following operators: get the parameters of an incoming request, an EXECI operator, write data to a channel. As described above for the operators of Fig. 27, instantiate and configure the corresponding operators. Before switching to the design of the platform model, we report in Fig. 29-Fig. 40, the activity diagrams of the other primitive components that constitute the model in Fig. 14. The user can draw the diagrams and interconnect the components as described so far for the single `Source` component.



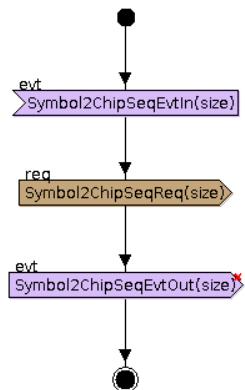
**Fig. 26.** The enumerated list of the buttons available to draw the activity diagram of a primitive component



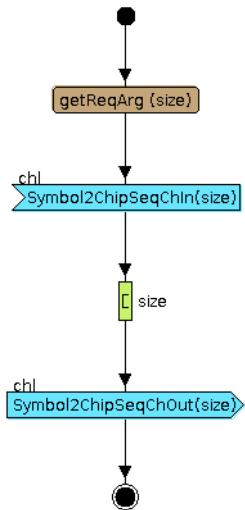
**Fig. 27.** The activity diagram of component F\_source



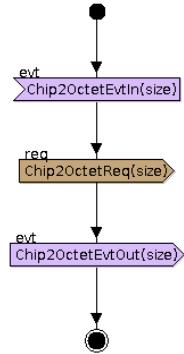
**Fig. 28.** The activity diagram of component `X_source`



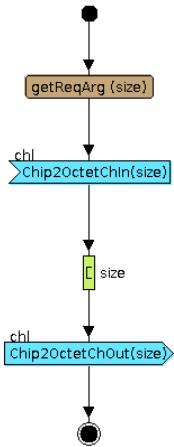
**Fig. 29.** The activity diagram of component `F_TXSymbol2ChipSeq`



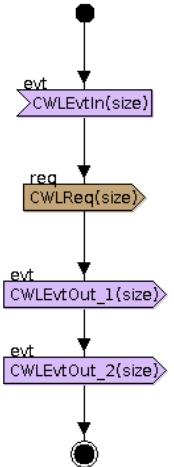
**Fig. 30.** The activity diagram of component `X_TXSymbol2ChipSeq`



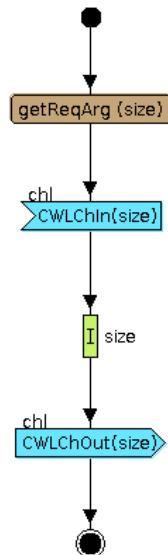
**Fig. 31.** The activity diagram of component F\_TXChip20ctet



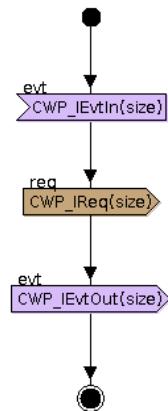
**Fig. 32.** The activity diagram of component X\_TXChip20ctet



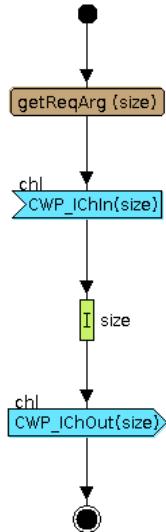
**Fig. 33.** The activity diagram of component F\_TXCWL



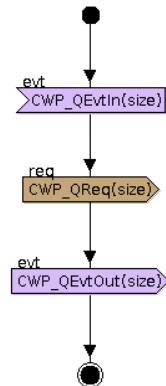
**Fig. 34.** The activity diagram of component X\_TXCWL



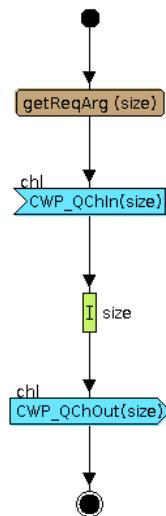
**Fig. 35.** The activity diagram of component F\_TXCWPI



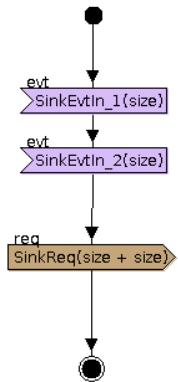
**Fig. 36.** The activity diagram of component X\_TXCWPI



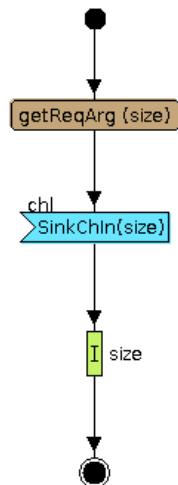
**Fig. 37.** The activity diagram of component F\_TXCWPQ



**Fig. 38.** The activity diagram of component X\_TXCWPQ



**Fig. 39.** The activity diagram of component F\_TXsink

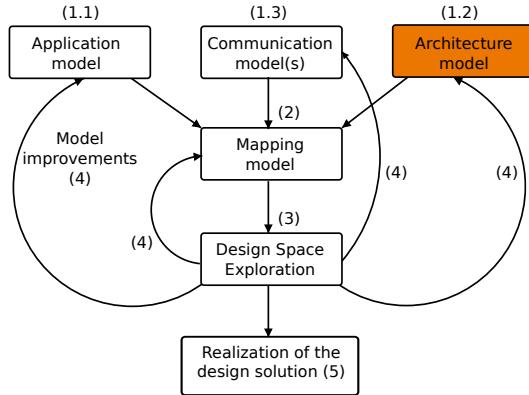


**Fig. 40.** The activity diagram of component X\_TXsink

### 7.3 Platform modeling

In this sub-section, we continue our design of the ZigBee transmitter (data-link layer) and discuss the model of the target hardware/software platform, Fig. 41.

The target platform for our case study is EMBB [17], a generic baseband architecture dedicated to signal processing



**Fig. 41.** The step of modeling the hardware/software platform, that is described in this section, in the context of the  $\Psi$ -chart design approach

applications.

Fig. 42a shows the UML Deployment Diagram of the EMBB architecture. EMBB is composed of a Digital Signal Processing part (DSP part) and a general purpose control processor (the main CPU). In the DSP part, left-hand side of Fig. 42a, samples coming from the air are processed in parallel by a distributed set of Digital Signal Processing Units (DSPU1 through DSPU $n$ ) interconnected by a crossbar (Crossbar). Fig. 42b illustrates the internal architecture of a DSPU: each unit is equipped with a local micro-controller ( $\mu$ C) that allows to reduce the intervention of the main CPU, a Processing Sub-System (PSS), a computational unit, and a Direct Memory Access controller (DMA) to transfer data in and out of the DSPU's local memory (the Memory Sub-System, MSS). The latter is mapped on the global address map of the main CPU and is accessible by the DMAs, the  $\mu$ Cs and the system interconnect. The system interconnect permits exchanges of control and data items: it is composed of a crossbar (Crossbar), a bridge (Main Bridge) and a main bus (Main Bus). The system interconnect is shared between the DSP part and the main CPU, where the control operations of an application are executed. The main CPU is in charge of configuring and controlling the processing operations performed by the DSPU $s$  and the data transfers. The main CPU has direct access to a memory unit (MAINmemory) and a bus interconnect (MAINbus) that communicates with the DSP part via the Main Bridge.

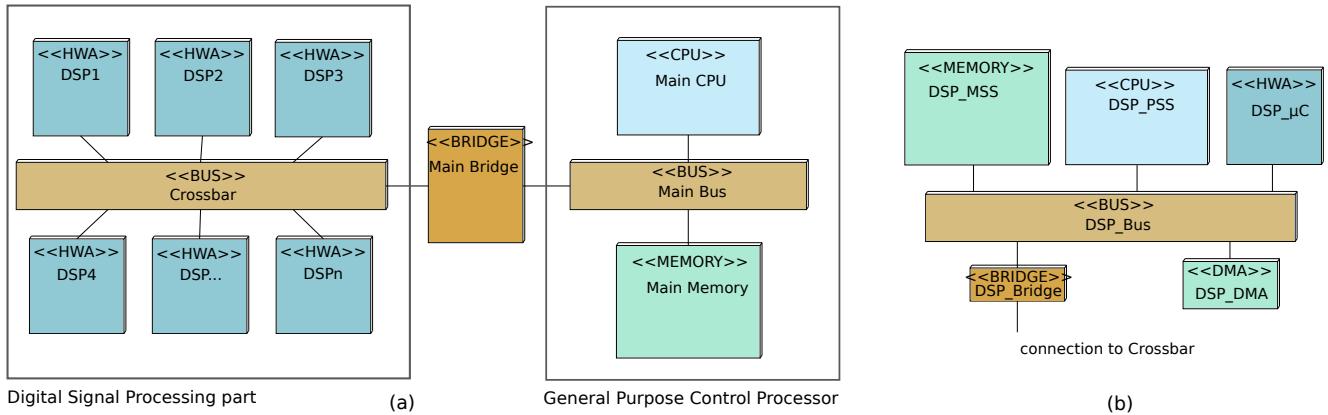
According to our design experience with the EMBB platform, the best configuration for signal-processing applications in terms of performance is the one with the following four DSP units:

- Front End Processor (FEP): it implements Discrete Fourier Transform and vector processing operations.
  - Interleaver (INTL): it implements permutations (i.e., interleaving and de-interleaving) of sequences of data samples.
  - Mapper (MAPPER): it transforms a frame of input symbols into a frame of complex numbers representing the points of a 2D constellation diagram, via Look-Up-Tables.
  - Analog to Digital-Digital to Analog Interface (ADAIF): a dispatcher that is capable of receiving up to 4 input streams from 4 A/D converters and of transmitting up to 4 output streams to 4 D/A converters.

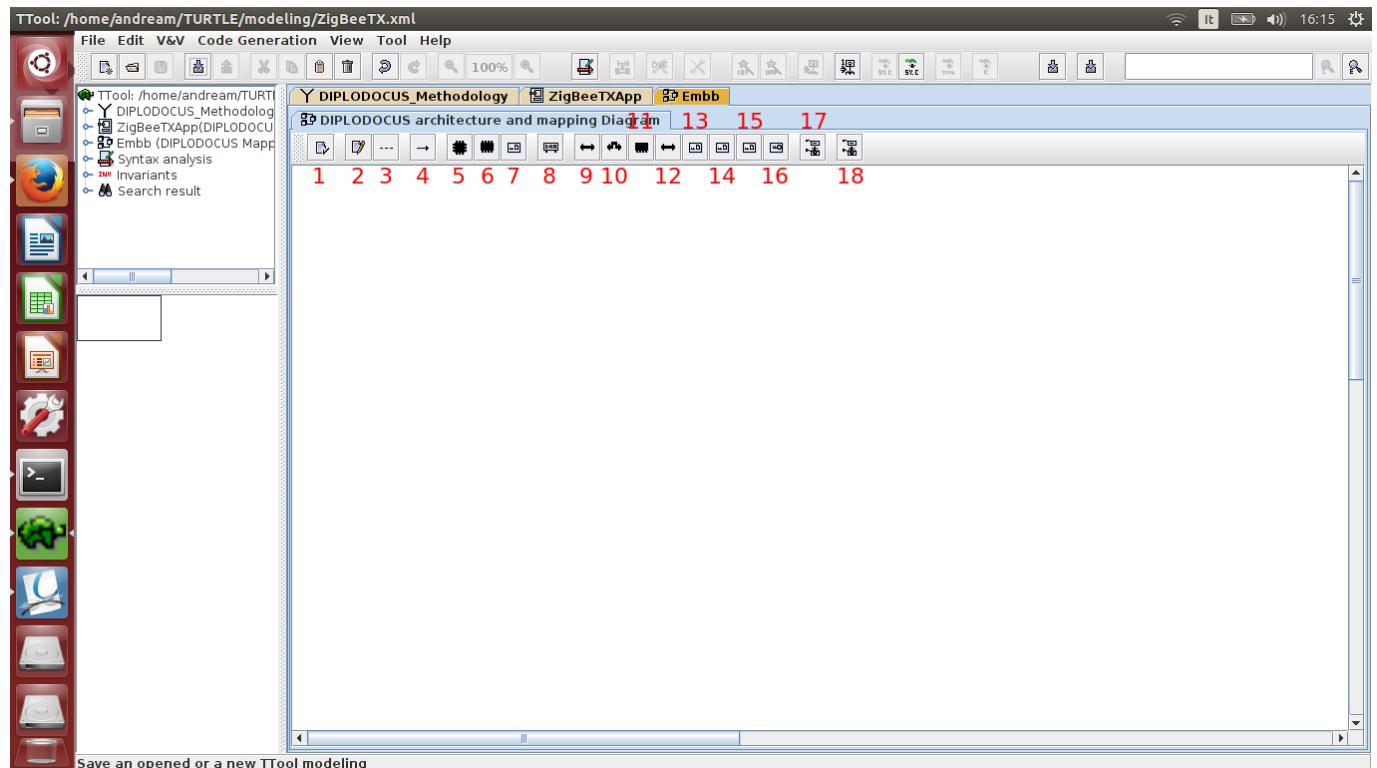
Therefore, the platform model that we will describe next is composed of the above DSP units.

## 7.4 Creating the platform model of EMBB

To create a platform model in TTool/DIPLODOCUS, right-click in the panel tab and select **New Partitioning - Architecture and Mapping**. This will open the panel in Fig. 43 where the buttons available to design a new model



**Fig. 42.** The UML Deployment Diagrams (architecture view) of an instance of EMBB, part (a), with its Digital Signal Processing part (left side) and main CPU (right side). Part (b) shows the internal architecture of each DSP unit.



**Fig. 43.** The design area for the platform model and the enumerated list of available buttons.

have been enumerated. As we did for the application panel, we rename the platform panel as EMBB (right click, then **Rename**). These buttons are described below:

1. Edit DIPLODOCUS Architecture Diagram
2. Add a comment: add a comment to the diagram
3. Comment connector
4. Connect two operators together: link two operators
5. CPU node: instantiate a node for a generic CPU block
6. Hardware accelerator node: instantiate a node for a generic hardware accelerator block
7. Map a task: map a primitive component from an application diagram onto a platform unit
8. DMA node: instantiate a node for a generic DMA block
9. Bus node: instantiate a node for a generic bus block
10. Bridge node: instantiate a node for a generic bridge block
11. Memory node: instantiate a node for a generic memory block
12. CP node: instantiate a node to map a Communication Pattern
13. Map an event/request: map an event/request onto platform unit
14. Map a channel: map a channel onto a memory unit, a bus or a bridge unit
15. Map a port: map a port onto the Communication Pattern mapping node
16. Map a key: it can be used to map an encryption/decryption key when the design also accounts for security properties, see Section 10.
17. Show/hide internal components: has no effect on this diagram
18. Show/hide DIPLODOCUS IDs: show or hide the identification (natural) numbers associated to each operator

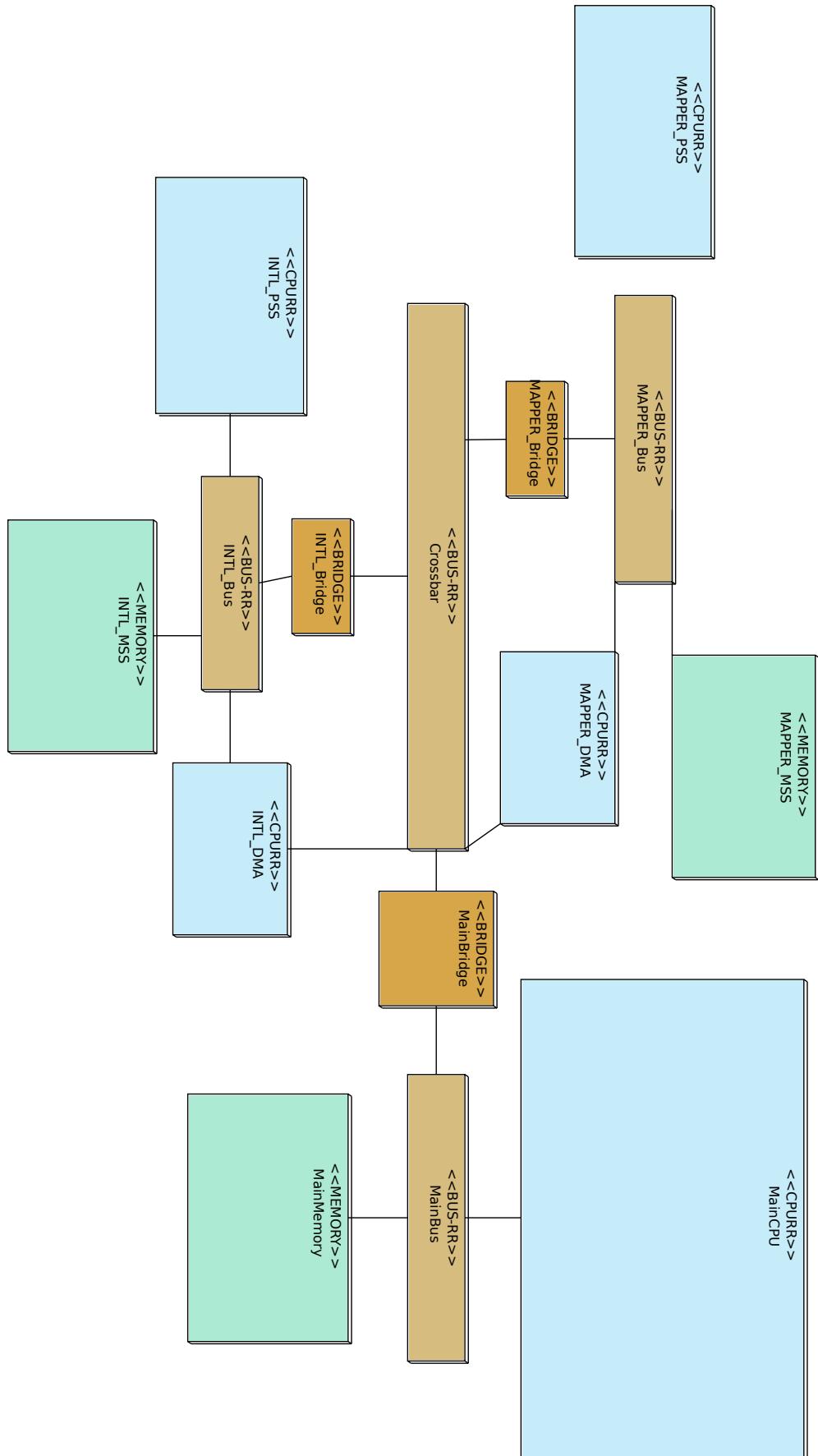
To draw the platform diagram of Fig. 42, the user must simply instantiate the desired units by clicking on the corresponding buttons, placing the units in the design area, interconnecting them, and assigning to each unit the correct performance parameters. With respect to Fig. 42b, each DSP unit is composed of a CPU block modeling the PSS, a memory block modeling the PSS's local memory. In this case study, we do not consider the micro-controllers of each DSP unit. Moreover, as the DMA units are not supported by the simulator, we model them as CPU blocks instead. Thus, a DSP unit results to be composed of 2 CPU blocks and a memory block interconnected by a local bus. The latter is then linked to the crossbar interconnect via a bridge. The crossbar interconnect itself is modeled as a bus. The control part of EMBB, left hand-side of Fig. 42a, is simply modeled as a generic CPU unit interconnected to a memory unit via a bus. This bus is itself connected to the crossbar interconnect via a bridge unit. Fig. 44 shows an excerpt of the platform model with the control part of EMBB on the right hand-side and two DSP units (Mapper and Interleaver) on the left hand-side.

In this design we do not use the local microcontrollers of each DSP unit, therefore our platform model does not include them. However, their modeling is very simple as it simply requires us to instantiate a CPU unit and to connect it to the internal bus of each DSP unit.

In the following tables we summarize the list of parameters that are associated to each generic platform unit and we provide a description of their meaning from the perspective of the simulation engine (Section 8). We specify here that parameters such as the pipeline size and the branching miss rate of a CPU impact the simulation time. For more details, please refer to Appendix 1.B.

In the following tables we list the numerical values of the performance parameters for each unit of the EMBB model that we have described so far.

In Table 11 the unit FEP\_PSS is assigned a different value for the parameter EXECI, as the FEP DSPU is capable of processing 2 input samples at a time. Among the performance characteristics of EMBB, the clock frequency of the main CPU and of the DSP units depends on the target implementation technology. On a Zedboard prototyping board[27], the main CPU runs at 650 MHz and each of the DSP units runs at 100 MHz. On the other hand, if EMBB were to be implemented in a dedicated integrated circuit (22 nm technology), both the DSP units and the main CPU would run at approximately 1GHz. Therefore, according to the target implementation technology the **Clock divider** must be assigned a proper value with respect to the **Master clock frequency**, Fig.45. The latter can be input to TTool/DIPLODOCUS from the dialog window that appears when checking the syntax of a mapping diagram. It is a parameter that is general to the whole design. The clock frequency of each active unit in the platform diagram (CPU, DMA, Hardware Accelerator) is computed by multiplying the Clock divider and the Master clock frequency. Therefore, in the case of EMBB being implemented in an integrated circuit, the Master clock frequency must be set to 1000 (MHz) and the Clock divider of each CPU unit must be set to 1. In the case of EMBB being implemented on a prototyping board, the Master clock frequency must be set to 100 (MHz), the Clock divider of the PSS of each DSP unit (modeled as a CPU block) must be set to 1 and the Clock divider of the main CPU must be set to 6. Only natural numbers can be assigned to the Clock divider parameter.



**Fig. 44.** An excerpt of the platform model of EMBB

**Table 1.** The performance parameters of a generic CPU unit

Parameter	Semantics
Arbitration policy	The arbitration policy used by the OS to schedule mapped tasks
Slice time (microseconds)	The maximum time allocated by the OS scheduler to execute a task
Number of cores	The number of cores of the CPU
Data size (bytes)	The size of an EXECI/EXECC operation, in number of bytes
Pipeline size (num. stages)	The number of stages of the pipeline
Task switching time (cycles)	The time taken by the OS for a context switch
Miss branching prediction (in %)	The miss percentage of the CPU branch prediction scheme
Cache miss (in %)	The percentage of cache misses
Go idle time (cycles)	The time taken by the OS and the CPU hardware to go idle
Max consecutive cycles before idle (cycles)	Number of consecutive cycles of NOPs before the CPU goes idle
EXECI (cycles)	The number of clock cycles corresponding to an integer operation
EXECC (cycles)	The number of clock cycles corresponding to an operation on complex numbers
Clock divider	This number defines the operating clock frequency of the CPU. It is expressed via a number that is used to divide the global design frequency, whose default value is 200 MHz. Thus a clock divider equal to 4 means that the CPU operates at $200/4 = 50$ MHz

**Table 2.** The performance parameters of a generic hardware accelerator unit

Parameter	Semantics
Data size (bytes)	The size of an EXECI/EXECC operation, in number of bytes
EXECI execution time (cycles)	The number of clock cycles expressed as an integer
Clock divider	This number defines the operating clock frequency of the CPU. It is expressed via a number that is used to divide the global design frequency, whose default value is 200 MHz. Thus a clock divider equal to 4 means that the CPU operates at $200/4 = 50$ MHz

**Table 3.** The performance parameters of a generic DMA unit

Parameter	Semantics
Data size (bytes)	Number of bytes that can be transferred in a clock cycle
Number of channels	The number of channels that can be used to transfer data independently
Clock divider	This number defines the operating clock frequency of the CPU. It is expressed via a number that is used to divide the global design frequency, whose default value is 200 MHz. Thus a clock divider equal to 4 means that the CPU operates at $200/4 = 50$ MHz

**Table 4.** The performance parameters of a generic bus unit

Parameter	Semantics
Arbitration policy	The arbitration policy used by the bus to schedule transactions
Data size (bytes)	Number of bytes that can be transferred in a clock cycle
Pipeline size (num. stages)	The number of stages of the pipeline
Slice time (in microseconds)	The maximum time allocated by the bus scheduler to execute a transaction
Clock divider	This number defines the operating clock frequency of the CPU. It is expressed via a number that is used to divide the global design frequency, whose default value is 200 MHz. Thus a clock divider equal to 4 means that the CPU operates at $200/4 = 50$ MHz
Bus privacy	See Section 10

**Table 5.** The performance parameters of a generic bridge unit

Parameter	Semantics
Buffer size (bytes)	The size of the internal buffer that temporarily stores data when the bridge has been allocated for another transaction
Clock divider	This number defines the operating clock frequency of the CPU. It is expressed via a number that is used to divide the global design frequency, whose default value is 200 MHz. Thus a clock divider equal to 4 means that the CPU operates at $200/4 = 50$ MHz

**Table 6.** The performance parameters of a generic memory unit

Parameter	Semantics
Data size (bytes)	Number of bytes in a single line of memory (size of a memory word)
Monitored	See Section 10
Clock divider	This number defines the operating clock frequency of the CPU. It is expressed via a number that is used to divide the global design frequency, whose default value is 200 MHz. Thus a clock divider equal to 4 means that the CPU operates at $200/4 = 50$ MHz

**Table 7.** The performance parameters for the bus units of each DSP unit

Platform units	Parameter	Value
Crossbar, ADAIIF_Bus, FEP_Bus, INTL_Bus, MAPPER_Bus	Arbitration policy	Round Robin
	Data size (bytes)	8
	Pipeline size (num. stages)	1
	Slice time <sup>5</sup> (microseconds)	10000
	Clock divider	1

**Table 8.** The performance parameters for the main Bus unit

Architecture unit	Parameter	Value
MainBus	Arbitration policy	Round Robin
	Data size (bytes)	4
	Pipeline size (num. stages)	1
	Slice time	10000
	Clock divider	1

**Table 9.** The performance parameters for the DMA units of each DSP unit (modeled as CPU units)

Platform units	Parameter	Value
FEP_DMA, ADAIIF_DMA, INTL_DMA, MAPPER_DMA	Scheduling policy	Round Robin
	Slice time (microseconds)	10000
	Number of cores	1
	Data size (bytes)	4
	Pipeline size (bytes)	5
	Task switching time (cycles)	20
	Miss branching prediction (in %)	2
	Cache miss (in %)	5
	Go idle time (cycles)	10
	Max consecutive cycles before idle (cycles)	10
	EXECI (cycles)	1
	EXECC (cycles)	1
	Clock divider	1

**Table 10.** The performance parameters for the (local) memory units of each DSP unit

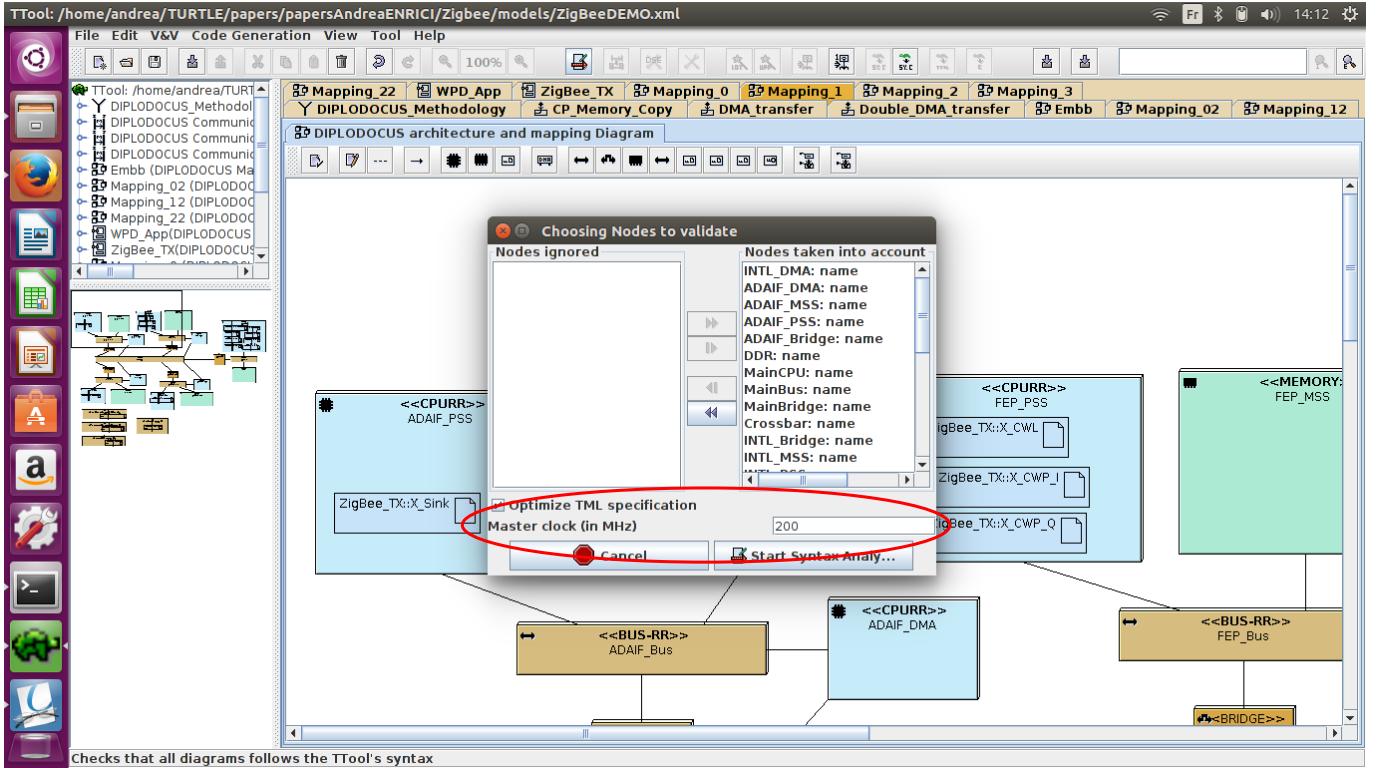
Platform units	Parameter	Value
ADAIIF_MSS, FEP_MSS, INTL_MSS, MAPPER_MSS, MainMemory	Data size (bytes)	4
	Clock divider	1

**Table 11.** The performance parameters for the PSS units (modeled as CPU units)

<b>Platform units</b>	<b>Parameter</b>	<b>Value</b>
ADAIF_PSS, INTL_PSS, MAPPER_PSS	Arbitration policy Slice time (microseconds) Number of cores Data size (bytes) Pipeline size (num. stages) Task switching time (cycles) Miss branching prediction (in %) Cache miss (in %) Go idle time (cycles) Max consecutive cycles before idle (cycles) EXECI (cycles) EXECC (cycles) Clock divider	Round Robin 10000 1 4 5 20 2 5 10 10 1 1 1
FEP_PSS	Arbitration policy Slice time (microseconds) Number of cores Data size (bytes) Pipeline size (num. stages) Task switching time (cycles) Miss branching prediction (in %) Cache miss (in %) Go idle time (cycles) Max consecutive cycles before idle (cycles) EXECI (cycles) EXECC (cycles) Clock divider	Round Robin 10000 1 4 5 20 2 5 10 10 2 1 1

**Table 12.** The performance parameters for the main CPU unit (control part of EMBB)

<b>Platform unit</b>	<b>Parameter</b>	<b>Value</b>
main CPU	Arbitration policy Slice time (microseconds) Number of cores Data size (bytes) Pipeline size (num. stages) Task switching time (cycles) Miss branching prediction (in %) Cache miss (in %) Go idle time (cycles) Max consecutive cycles before idle (cycles) EXECI (cycles) EXECC (cycles) Clock divider	Round Robin 10000 2 4 5 20 2 5 10 10 1 1 1



**Fig. 45.** The Master clock frequency can be input from the syntax analysis window of a mapping diagram

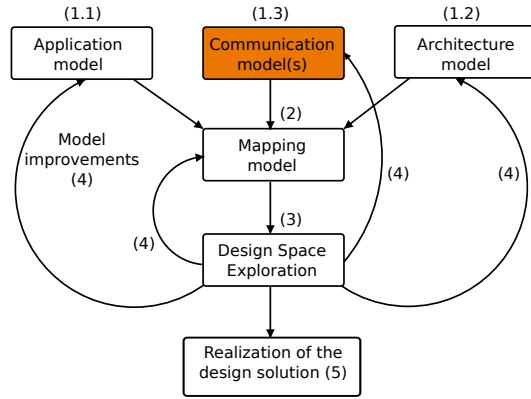
## 7.5 Communication protocols and patterns modeling

The introduction of separate models dedicated to designing communication protocols and patterns in the Y-chart approach, Fig. 1, results into the  $\Psi$  chart approach, Fig. 2. This introduction aims at solving the *communication mismatch* issue. The latter occurs when mapping communications in the Y-chart (step 2 in Fig. 1), due to a *mismatch* between the description of communications contained in the application and in the architecture models. This mismatch involves, on one hand, the primitives and operational semantics used to describe communications in the application model (e.g., point-to-point data channels with blocking read()/write() operations) and, on the other hand, the primitives and operational semantics used in the platform model (e.g., configuring and executing a DMA data transfer or a series of bus transactions).

The communication models of the  $\Psi$ -chart that we implemented for TTool/DIPLODOCUS are called Communication Patterns (CPs). These models describe communication protocols at the data-link layer of the ISO/OSI reference model [12]. CPs are deployed to model communication protocols at Electronic System-Level of abstraction. Therefore, CPs do not consider the effects of caching on communications (e.g., the transfers between a cache and main memory due to a cache miss) that occur at a micro-architecture level of abstraction. Caching effects are abstracted in the timing attributes of a generic CPU block in the platform model. An attribute, called *cache-miss ratio*, is used by the DSE engine of TTool/DIPLODOCUS as a penalty that is associated to each read/write operation between generic CPUs and memory blocks of the platform model.

A Communication Pattern describes the *behavior* of a communication protocol, intended as a set of rules for the exchange of data between *components* of an embedded system. A component is intended as a generic architecture unit, regardless its implementation, i.e., hardware, software or both.

According to the communication services offered by EMBB, information can be transferred via bus transactions, DMA transactions or via CPU load/store operations (CPU memory-copy). In the next paragraphs, we show the models that we created for these transfer mechanisms. We remind to the reader that a CP is denoted with a main SysML Activity Diagram that describes the algorithm of a communication protocol/pattern. This diagram, in turn, references other SysML Activity Diagrams or UML Sequence Diagrams. The UML Sequence Diagram for a CP is used to model interactions (e.g., signals) between actors of a communication protocol (e.g., master, slave). These actors are represented as UML instances. Each instance has a set of unique attributes (e.g., integer, boolean variables) and can exchange synchronous parameterized messages with other instances. Instances belong to 3 different types: transfer, control and



**Fig. 46.** The communication model step, that is described in this section, in the context of the  $\Psi$ -chart design approach

storage. Transfer instances (e.g., they denote a bus) can only forward messages, storage instances (e.g., they denote a memory) can only receive messages and control instances (e.g., they denote a CPU) can both send and receive messages.

For the interested reader, Appendix 1.A provides a more formal description of Communication Patterns and of all their properties and semantics.

Last but not least, TTool cannot currently analyze models of communication patterns. As a consequence, **communication patterns can be used for documentation purpose only**. Yet, TTool internally understands three CPs: "DMA", "double DMA" and "MemoryCopy". The identification of these patterns is made according to their NAMING, as given in the code of the tmltranslator.TMLCPLib.java source file:

```

1 public boolean isDMATransfer() {
2     return typeName.compareTo("DMA_transfer") == 0;
3 }
4
5 public boolean isDoubleDMATransfer() {
6     return typeName.compareTo("Double_DMA_transfer") == 0;
7 }
8
9 public boolean isMemoryCopy() {
10    return typeName.compareTo("CP_Memory_Copy") == 0;
11 }

```

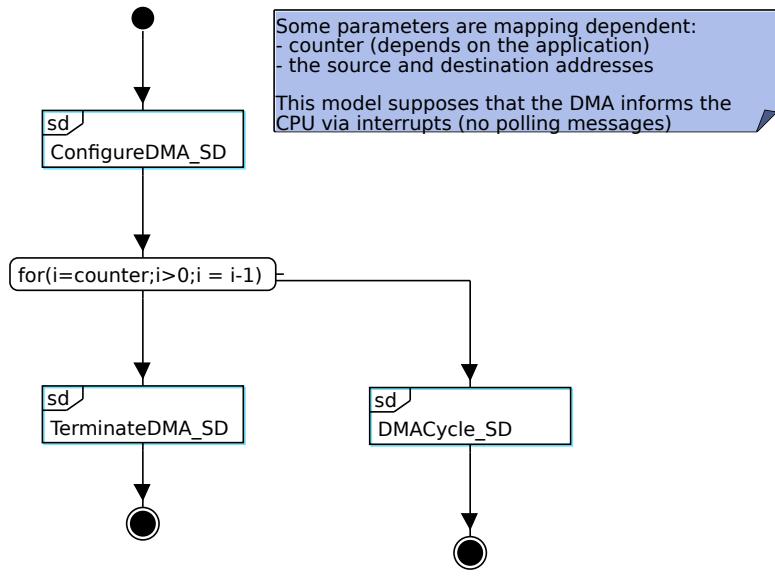
Finally, whatever the content of the communication pattern, what is currently relevant for TTool is the name of the communication pattern.

## 7.6 Modeling a DMA data transfer with Communication Patterns

In this subsection we illustrate how our Communication Patterns can be deployed to model the communication protocol of a generic data transfer via DMA. We first show the Communication Pattern modeling an interrupt-based DMA transfer, where the transfer completion is signaled by the DMA controller with an interrupt to the CPU. Next, we present a Communication Pattern where the CPU polls the DMA controller to detect the transfer completion.

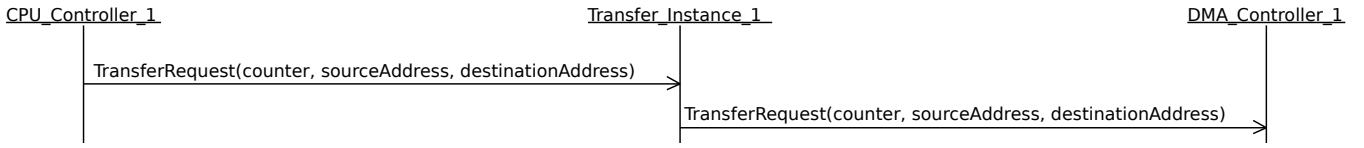
The main Activity Diagram of the Communication Pattern for the interrupt-based transfer is illustrated in Fig. 47. In this diagram we decomposed the communication protocol into three steps (Sequence Diagrams): first the data transfer is configured (ConfigureDMA\_SD in Fig. 47), then data are iteratively transferred (DMACycle\_SD in Fig. 47) and the data transfer is terminated (TerminateDMA\_SD in Fig. 47). Data are transferred iteratively, as expressed by the *iteration* operator, based on the value assigned to the control variable *counter* during the initialization phase.

The Sequence Diagram ConfigureDMA\_SD is depicted in Fig. 48. Here, we model how a generic CPU unit configures the DMA controller for a data transfer. These two units are represented as two instances of type controller, interconnected by an instance of type transfer. The CPU instance sends the source and destination addresses (*sourceAddress*,



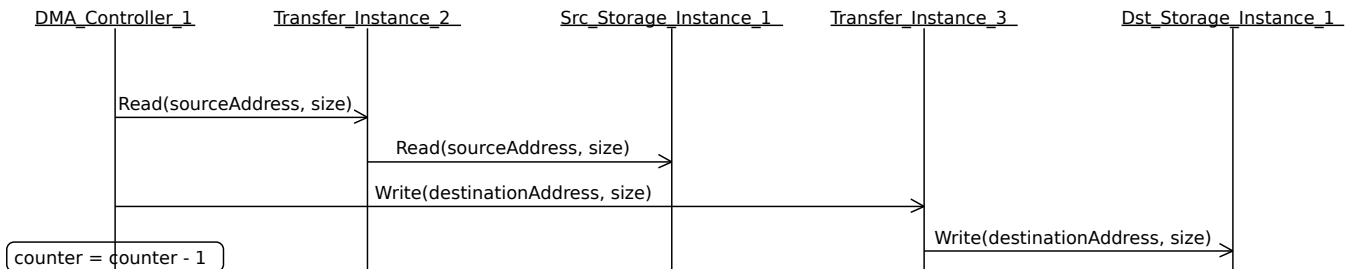
**Fig. 47.** The main Activity Diagram of a Communication Pattern modeling a data transfer via DMA. In this case, interrupts are used as a communication mechanism to notify the transfer completion.

`destinationAddress`) and the amount of data to transfer (`counter`) as parameters of the message `TransferRequest()` to the DMA controller, via the transfer instance. The DMA controller, upon reception of the message, retrieves the parameters and sets an internal counter (`counter`) whose value is equal to the amount of data to transfer. At this point, it is important to remark that all message parameters are not initialized as our Communication Patterns are independent of the application model. It is in fact the application model that expresses the numerical value of the exact amount of data that must be transferred. Similarly, the exact value of the source and destination addresses depends on the units of the platform model where the CP is mapped. This missing information will be added to the models only during the mapping phase.



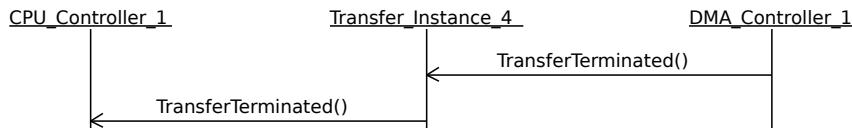
**Fig. 48.** The Sequence Diagram ConfigureDMA\_SD of Fig. 47.

In the Sequence Diagram `DAMCycle_SD` of Fig. 49, we model one DMA transfer cycle. For this purpose we instantiate the DMA controller of Fig. 48, a source and destination storage instances and two transfer instances. The latter are used to interconnect the DMA controller to the two storage instances. The transfer cycle is modeled with the DMA controller reading samples out of the source storage component via a parametrized `Read()` message, where `size` indicates the amount of data that is read. Next, the DMA controller writes data to the destination storage via a `Write()` message and decrements the counter. Similarly to the above diagrams, we note here that parameters `size`, `sourceAddress` and `destinationAddress` depend on the system's architecture and are therefore left as unspecified. They will be assigned a value after the instances are mapped onto a specific DMA and memory units as detailed in subsection 7.8. The message parameter `size` defines the amount of data that the channel of the DMA controller is able to transfer at each transfer-cycle.



**Fig. 49.** The Sequence Diagram DMACycle\_SD of Fig. 47.

In the last Sequence Diagram TerminateDMA\_SD of Fig. 50, the DMA controller sends an acknowledgment message (`TransferTerminated()`) to the CPU controller to inform the latter that the transfer has terminated. One last point that we want to highlight is that all the above-mentioned transfer instances (i.e., `TransferInstance_1-4`) are different objects of the same instance class (i.e., *transfer*). As a matter of fact, at this level of abstraction we have no information about the physical route that is used to transfer information in the target platform.



**Fig. 50.** The Sequence Diagram TerminateDMA\_SD of Fig. 47.

Fig. 51 shows the main Activity Diagram of a Communication Pattern where the CPU polls the DMA controller in order to detect the termination of the data transfer. In this case, we modeled the communication protocol with one Sequence Diagram, ConfigureTransfer and two Activity Diagrams: TransferCycleAD and PollingCycleAD. Similarly to the Communication Pattern of Fig. 47, we first model the configuration of the data transfer with a Sequence Diagram, ConfigureTransfer. Next, we model the transfer cycle and the polling cycle being executed in parallel. These two phases cannot be modeled directly with Sequence Diagrams as they require the description of a set of interactions that are iteratively executed.

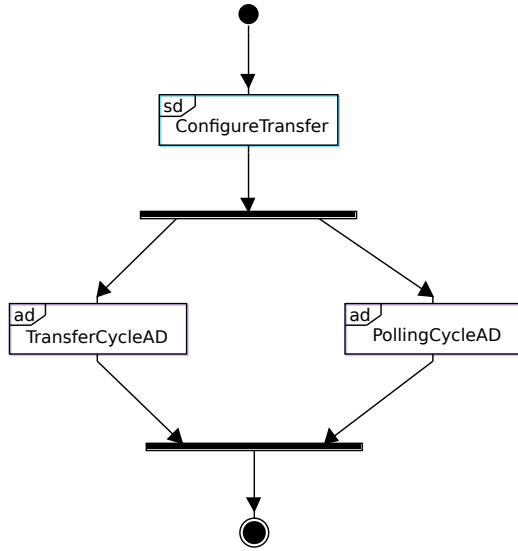
The Sequence Diagram ConfigureTransfer of Fig. 52 is similar to the one described in Fig. 48: the only difference is the initialization of the flag `transferTerminated` to false. The latter is set to true by the DMA controller upon termination of the data transfer.

Fig. 53 depicts the Activity Diagram TransferCycleAD where the Sequence Diagram DMACycle\_SD is iteratively executed as long as the control variable `counter` is greater than zero, meaning that there are still data to be transferred. Once all data have been transferred, the DMA controller sets the flag `transferTerminated` to true in Sequence Diagram EnableFlag. Diagram DMACycle\_SD is not illustrated here as it is the same as diagram DMACycle\_SD of Fig. 49.

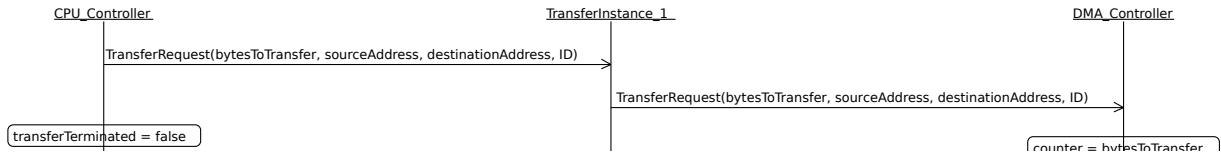
Fig. 54 shows the Activity Diagram PollingCycleAD, where Sequence Diagram PollingCycleSD models the iterative polling of the DMA controller. This scenario is depicted in Fig. 55 where the CPU intermittently polls the state of the data transfer via a `PollingRequest()` message to the DMA controller until the message parameter `transferTerminated` is set to true. The exact value of variable `waiting_time` is left as unspecified as it depends on the mapping.

## 7.7 Communication models

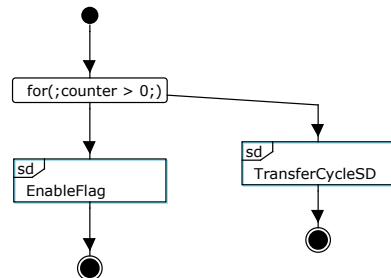
Given the communication resources and services that are available in EMBB, the communications protocols that we need to model are based on DMA transfers with interrupt mechanisms. The models for a DMA transfer have already been illustrated in the previous paragraph and therefore we will not repeat their diagrams here.



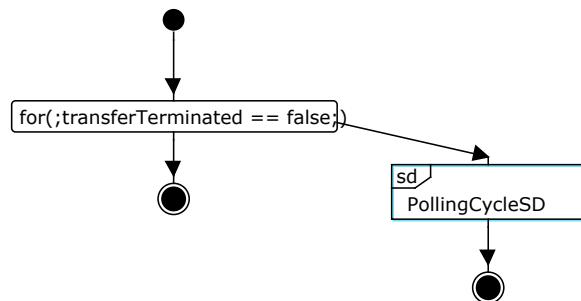
**Fig. 51.** The main Activity Diagram of the Communication Pattern modeling the data transfer via DMA. In this case, polling is used as a mechanism to notify the transfer termination.



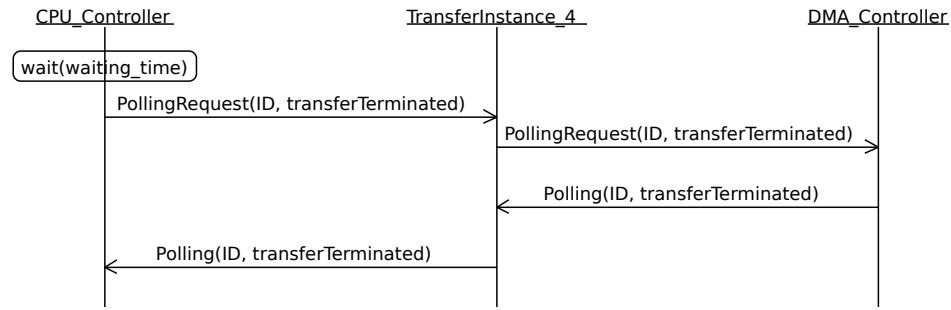
**Fig. 52.** The Sequence Diagram ConfigureTransfer of Fig. 51.



**Fig. 53.** The Activity Diagram TransferCycleAD of Fig. 51.



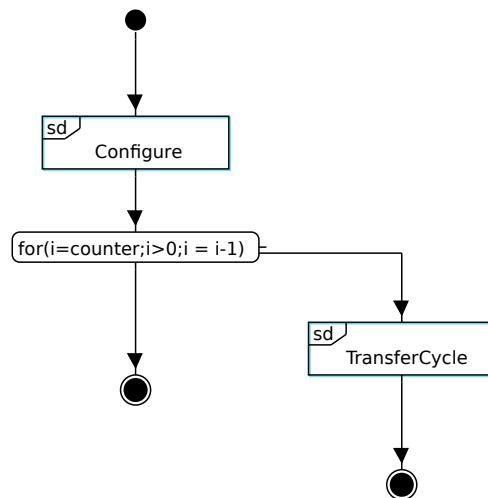
**Fig. 54.** The Activity Diagram PollingCycleAD of Fig. 51.



**Fig. 55.** The Sequence Diagram PollingCycleSD describing the message exchanges of the polling cycle of Fig. 54.

With respect to these models, another novel Communication Pattern that we need for the ZigBee transmitter is the one depicted in Fig. 56. Here, the main Activity Diagram captures a memory copy transfer. This transfer is used in the context of EMBB to model a data transfer from the MainMemory to the local memory of any DSP unit, via a store operation issued by the MainCPU. Fig. 57 displays the Activity Diagram TransferCycle, which models the message exchanges similarly to the diagram in Fig. 49. Attribute numData specifies the number of data that are transferred at each cycle. Another novel Communication Pattern that we used in our case study is the one illustrated in Fig. 58. This CP captures a pair of sequential DMA transfers and can be used to model a copy operation from one source storage to two different destination storages.

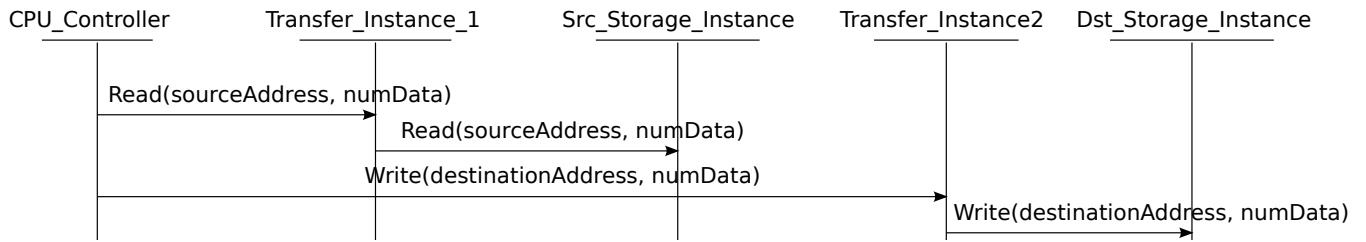
The main Activity Diagram of this Communication Pattern is simply composed of two references to Activity Diagrams, composed via the sequence operator. These references point to the Activity Diagrams of Fig. 59 and 60 that describe a standard DMA transfer as the one presented above in Fig. 61-Fig. 66.



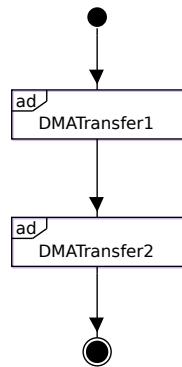
**Fig. 56.** The Communication Pattern for a memory copy data-transfer

**The communication mismatch in EMBB** In the application and platform models that we presented so far, we remind the reader that communications are described in terms of:

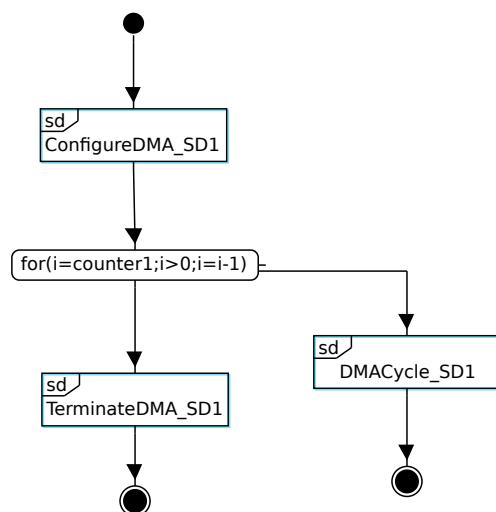
- Point-to-point unidirectional data-channels between tasks of the application model. A task accesses these channels via: (i) blocking read/blocking write, (ii) blocking read/non-blocking write, (iii) non-blocking read/non-blocking write operations.



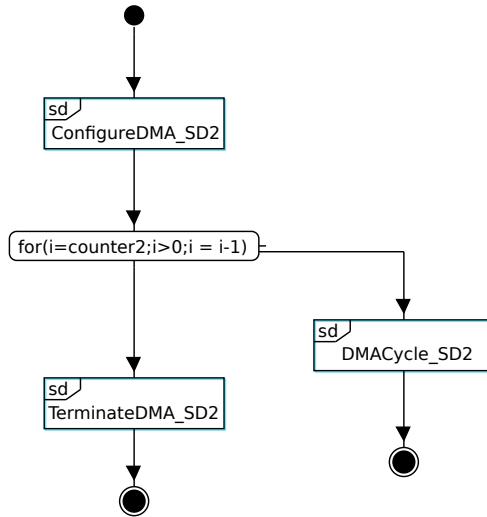
**Fig. 57.** The Sequence Diagram TransferCycle referenced in Fig. 56



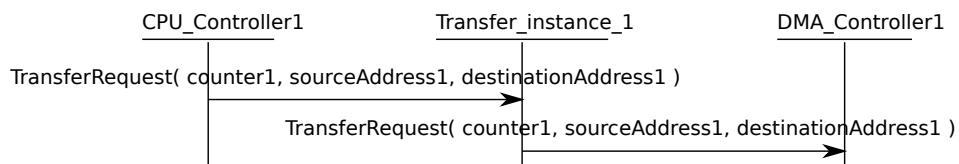
**Fig. 58.** The Communication Pattern for a pair of sequential DMA transfers



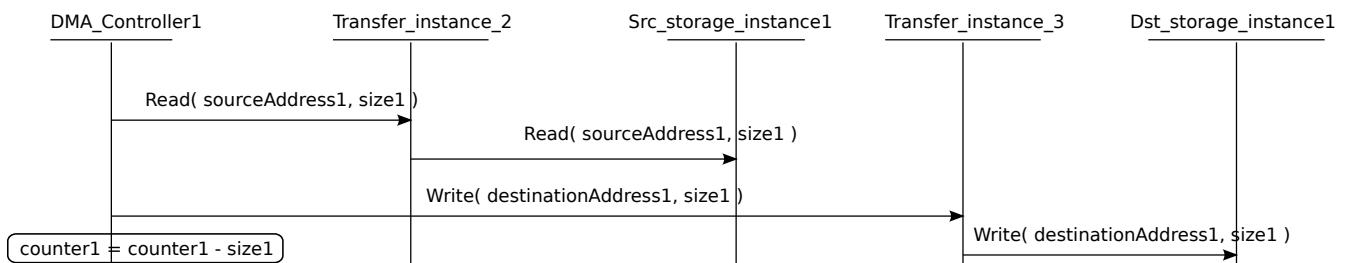
**Fig. 59.** The Activity Diagram referenced by DMATransfer1 in Fig. 58



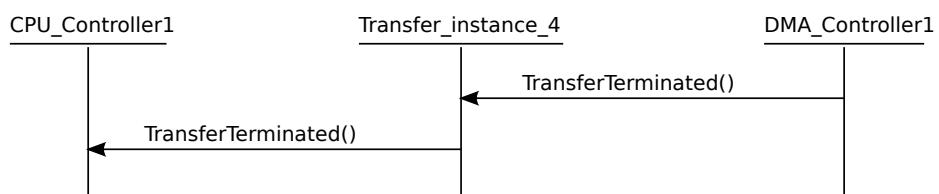
**Fig. 60.** The Activity Diagram referenced by DMATransfer2 in Fig. 58



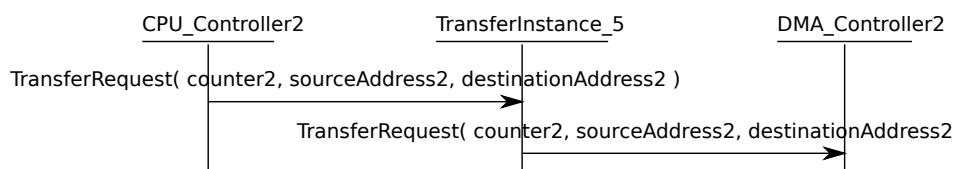
**Fig. 61.** The Sequence Diagram ConfigureDMA\_SD1 of Fig. 59.



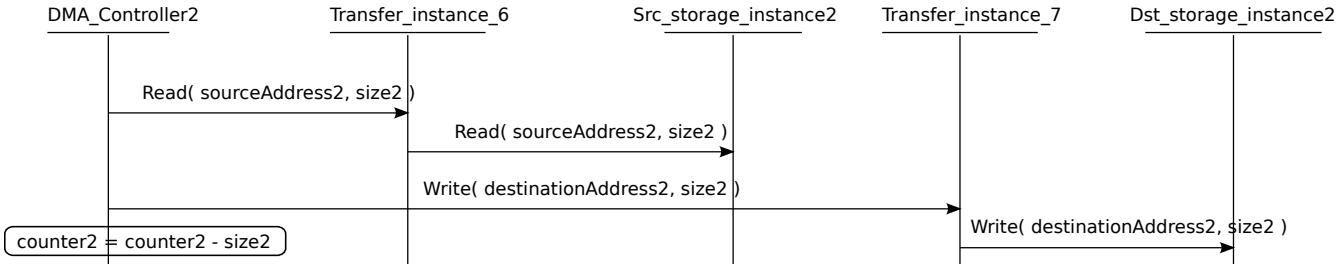
**Fig. 62.** The Sequence Diagram DMACycle\_SD1 of Fig. 59.



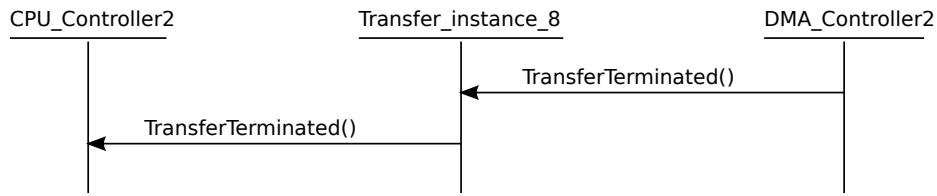
**Fig. 63.** The Sequence Diagram TerminateDMA\_SD1 of Fig. 59.



**Fig. 64.** The Sequence Diagram ConfigureDMA\_SD2 of Fig. 60.



**Fig. 65.** The Sequence Diagram DMACycle\_SD2 of Fig. 60.



**Fig. 66.** The Sequence Diagram TerminateDMA\_DMA2 of Fig. 60.

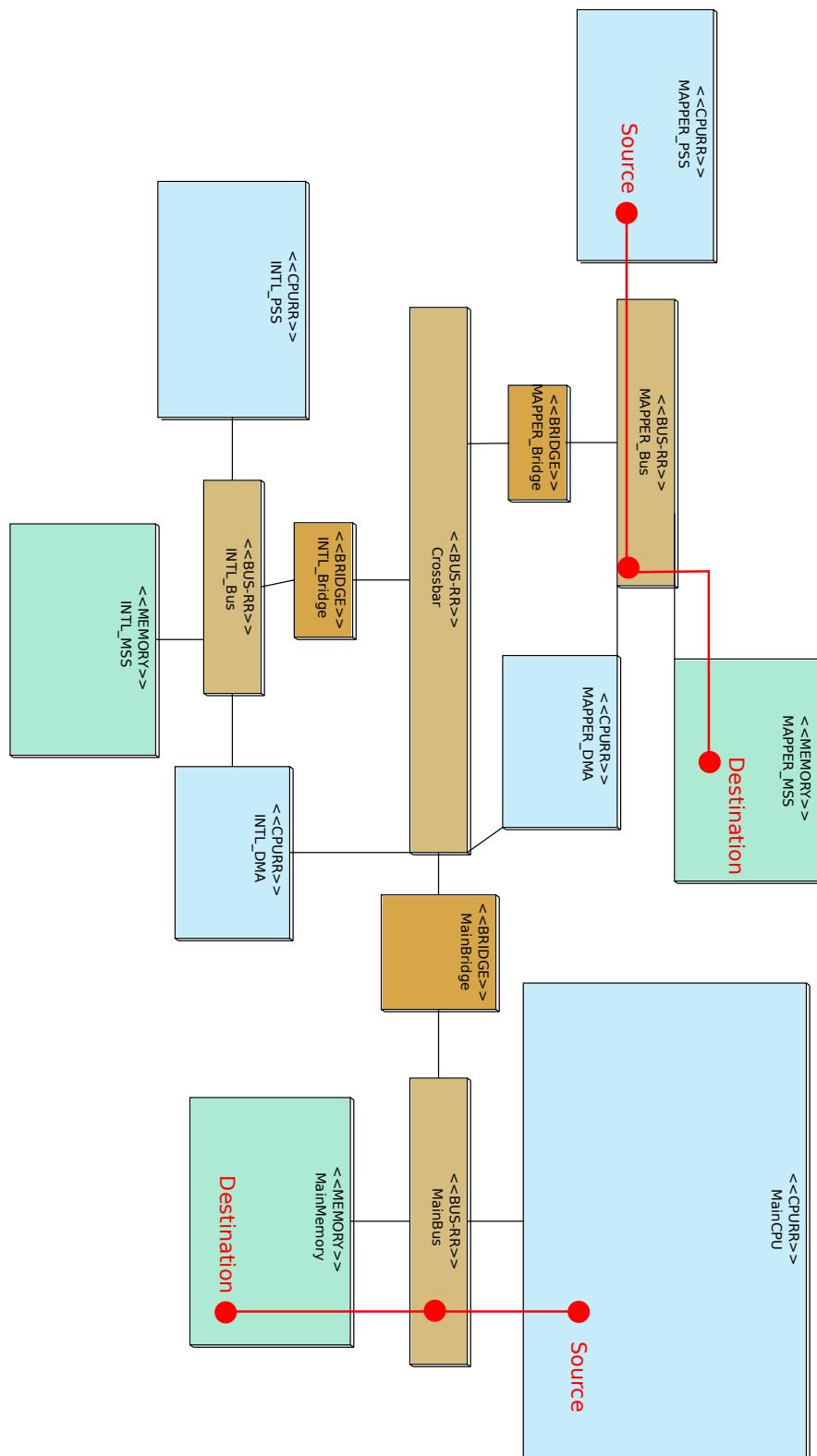
- Read/write operations performed by processing units (i.e., CPU, Hardware Accelerator, HwA) to/from memory units in the architecture model. CPU units and HwA units issue read and write requests over bus and bridge units. As described in [4], buses may be endowed with several independent communication channels which can be used simultaneously. The communication between architecture units makes use of the circuit switching paradigm: during the entire data transmissions on more than one bus, at least one communication channel on all involved buses has to be available and is reserved. Reservation is accomplished starting from the CPU towards the memory element in causal fashion.

Given the semantics of an application and platform models in TTool/DIPLODOCUS, communication mismatches arise when data are transferred via paths in the platform model that encompass a sequence of more than one bus between a source CPU and a destination memory.

Specifically to the case study described in this tutorial, communication mismatches arise when data are transferred:

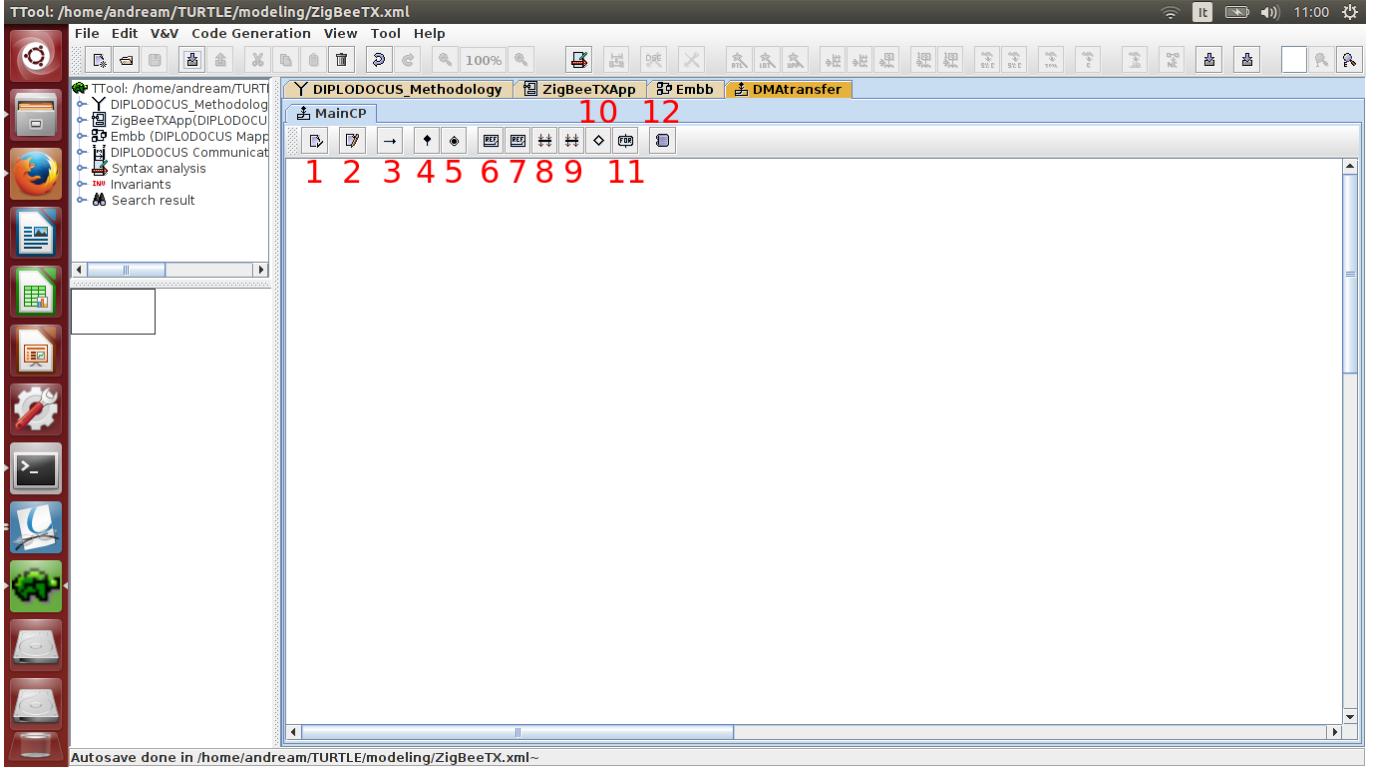
- from MainMemory to any of the DSP local memories and vice-versa, e.g., path DDR-MainBus-MainBridge-Crossbar-MAPPER\_Bridge-MAPPER\_Bus-MAPPER\_MSS in Fig. 67
- from a DSP local memory to any other DSP local memory, e.g., path MAPPER\_MSS-MAPPER\_Bus-MAPPER\_Bridge-Crossbar-INTL\_Bridge-INTL\_Bus-INTL\_MSS in Fig. 67

The only case in which there is a communication *match* between the application and the platform is given by the path MainCPU-MainBus-MainMemory or the path that links a DSP PSS to its local memory, e.g., MAPPER\_PSS-MAPPER\_Bus-MAPPER\_MSS in Fig. 67.



**Fig. 67.** An excerpt of the platform model of EMBB. In red are highlighted two paths processor-bus-memory that are NOT source of a communication mismatch when mapping data channels from the application model.

**Creating Communication Pattern diagrams** To create a Communication Pattern, left click on the panel tab and select New Partitioning – Communication Pattern, then rename the panel to DMAtransfer. Fig. 68 shows the design window of the main Activity Diagram of a Communication Pattern and enumerates the available buttons.



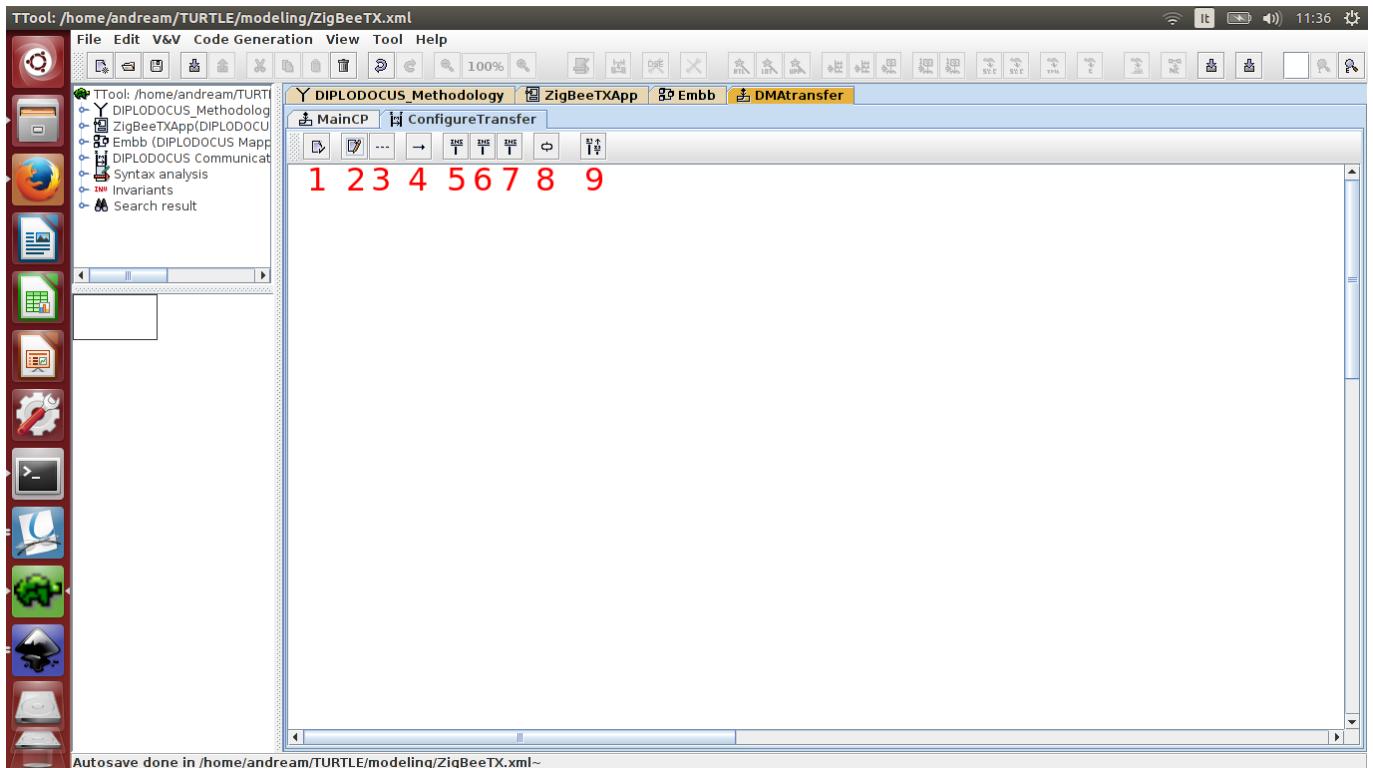
**Fig. 68.** The design window for an Activity Diagram of a Communication Pattern

1. Edit Communication Pattern Diagram
2. Add a comment: add a comment to the diagram
3. Connector: add a connector between two elements of a diagram
4. Start: instantiate the start node
5. Stop: instantiate the stop node
6. Reference to a SD: instantiate a reference to CP Sequence Diagram
7. Reference to an AD: instantiate a reference to a CP Activity Diagram
8. Fork: instantiate the fork operator
9. Join: instantiate the join operator
10. Choice: instantiate the choice operator
11. Loop: instantiate a for-loop operator
12. Enhance

The design window that is automatically created is that of the main Activity Diagram, that is the top-most diagram of a Communication Pattern. To create an Activity Diagram, that will be referenced by the main diagram, instantiate a reference to an Activity Diagram (button n.7), right-click on the reference and select **Create Activity Diagram**. This will automatically create a diagram whose design window is the same as the one of the main Activity Diagram. To design the main Activity Diagram of the DMA transfer, as illustrated in Fig. 47, we must instantiate and interconnect 1 start node, 3 references to Sequence Diagrams, 1 for-loop and 2 stop nodes. For this purpose, simply click on the corresponding buttons, then place and connect the elements in the design area. To rename a reference to a Sequence Diagram, double click on it and enter the desired name. To configure the initialization, the termination and the increment conditions of the for-loop also double click on the for-loop and enter the desired parameters.

To design diagrams **ConfigureTransfer**, Fig. 48, **TransferCycle**, Fig. 49, and **TerminateDMA\_SD**, Fig. 50, right-click on their references, select **Open diagram**.

Fig. 69 shows the design window of a CP Sequence Diagram and enumerates the available buttons.



**Fig. 69.** The design window for a CP Sequence Diagram

1. Edit the Sequence Diagram
2. Add a comment: add a comment to the diagram
3. Comment connector
4. Asynchronous message: instantiate an asynchronous message between a sender and a receiver instances
5. Storage instance: instantiate an instance of type storage
6. Controller instance: instantiate an instance of type controller
7. Transfer instance: instantiate an instance of type transfer
8. Action state: instantiate an action state for an instance
9. Align instances: horizontally align multiple instances

To design diagram **ConfigureTransfer**, instantiate 2 controller instances and 1 transfer instance by clicking on button n.6 and button n.7 (Fig. 69), respectively. Double click on an instance to open the window that allows us to rename it and to add attributes. Rename the initiator instance as **CPU\_Controller**, then add attributes **counter** of type **Natural**, **sourceAddress** and **destinationAddress** of type **Address**. Similarly, rename the receiver controller instance as **DMA\_Controller** and the transfer instance as **TransferInstance\_1**. Before including messages to the diagram, add to these 2 instances the same attributes as those for the **CPU\_Controller**. This is important because, given a pair of instances of a Communication Pattern,  $I_1, I_2$ , interconnected by message  $M$  that carries parameters  $P = (p_1, p_2, \dots, p_n)$ , the user must define parameters  $P$  for both  $I_1$  and  $I_2$ .

Now, connect the instances with asynchronous message **TransferRequest** as shown in Fig. 48. Click on button n. 4 (Fig. 69) and the tool will highlight (yellow boxes) the connection points along the lifelines of the available instances. Click on one of these points on a sender instance and then click on a connection point on a receiver instance. Double click on the message arrow, enter the message name and its parameters.

To design diagram **TransferCycle**, instantiate 1 controller instance (named **DMA\_Controller**), 2 transfer instances (named **TransferInstance\_2** and **TransferInstance\_3**) and 1 storage instance (named **SOURCE\_Storage**). Then include messages **Read()** and **Write()**, Fig. 49, as described above for diagram **ConfigureTransfer**. To add the action state that decrements attribute **counter**, click on button n.8 (Fig. 69) and place the action state on a target instance. Double click the action to enter the action that decrements the counter: **counter = counter - size**.

To design diagram **TerminateDMA\_SD** instantiate 2 controller instances (named **DMA\_Controller** and **CPU\_Controller**) and 1 transfer instance (named **TransferInstance\_4**). Then interconnect them via message **TransferTerminated()**

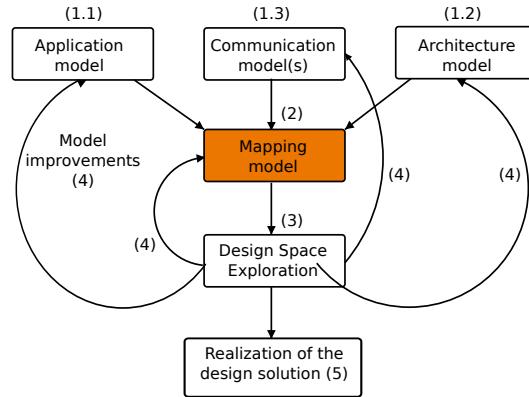
as shown in Fig. 50.

To design the Communication Patterns for two serial DMA transfers as in Fig. 58, create a new CP. Add the references to 2 Activity Diagrams, button n.7 in Fig. 68, and a stop node, button n.5 in Fig. 68. Then, simply name the two references **DMATransfer1** and **DMATransfer2**. For each DMA transfer, copy the activity and sequence diagrams that we have just created. Then, assign unique names to the diagrams, the instances and their attributes as shown in Fig. 59 - Fig. 66.

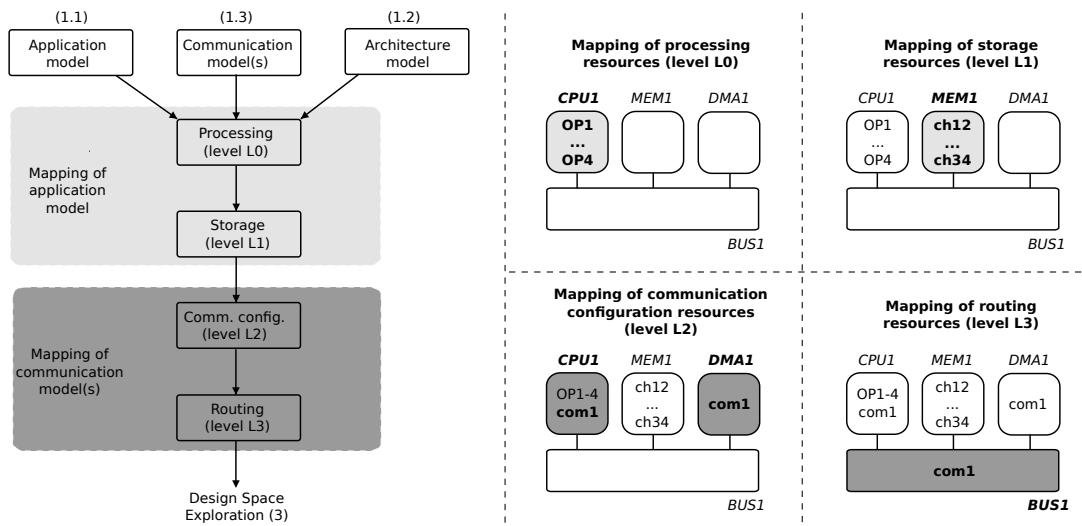
After this point of the tutorial, we let the user design by himself/herself the diagrams for the CPU memory-copy operation of Fig. 56.

## 7.8 Mapping

Fig. 71 details the methodology that we defined in the  $\Psi$ -chart approach (Fig. 70) in order to create a complete mapping model (step 2 in Fig. 2). The mapping methodology in Fig. 71 shows the process required to *bind* the application and communication models onto the platform model.



**Fig. 70.** The mapping step that is described in this section, in the context of the  $\Psi$ -chart design approach



**Fig. 71.** The mapping methodology of the  $\Psi$ -chart (left side) and the models for each step (right side).

**Computation (level L0).** The computational parts of the application model are mapped to the platform model. For instance, a node in the application MoC that models an FFT operation is mapped to a Digital Signal Processor (DSP) unit; a node modeling a control task is mapped to a Central Processing Unit (CPU). Similarly, for variables which are dependent on the specific characteristics of the mapped unit, a value is assigned accordingly. For instance, variables describing abstract data types (e.g., complex numbers) are assigned a value (e.g., `cpx32` in case the mapped DSP unit represents complex numbers with 16 bits for the imaginary part and 16 bits for the real part).

**Storage (level L1).** Any behavior and variable in the application model that is related to the storage of data or control information is mapped. A system engineer selects the platform units (e.g., memories, buffers) that will store

the data and/or control information produced or consumed by the computations mapped at level L0. According to the selected units and their characteristics, parameters are assigned a value, e.g., the size of a buffer.

At this point, data dependencies in the application model must be associated to the communication models that describe the corresponding transfer of data. In our implementation of the  $\Psi$ -chart in TTool/DIPLODOCUS this is performed by the user who associates a data channel between computations to a Communication Pattern (subsection 7.5). We specify that this solution is not imposed by the design principles of the  $\Psi$ -chart approach. Other types of relations may be deployed according to the characteristics of the specific design tool into which the  $\Psi$ -chart is implemented (e.g., matching signature operations).

**Communication configuration (level L2).** The behavior and parameters of a communication model are mapped to the platform model. A system engineer selects the platform units that will be in charge of configuring the data transfers that move data from the source to the destination storage units mapped at level L1.

**Routing (level L3).** The route that data will take to be transferred between a source and a destination storage is chosen in terms of transfer units (e.g., bus, bridge) according to the topology of the architecture model.

We describe here the mapping of the application and communication models of the ZigBee transmitter, according to the methodology in Fig. 71.

We first map the computations (level L0) of the application model. As mentioned above, a mapping model is based on an instance of the platform model. To clone the platform model, right-click on the platform panel, select **Clone** then rename the newly copied diagram as **Mapping**. The mapping of computations is summarized in Table 13: each control task is mapped on the Main CPU unit and the data-processing tasks are mapped on the DSPUs. In a design where also the microcontrollers of each DSP unit are deployed, an alternative mapping would be to delegate some of the control tasks to these local microcontrollers. To add the mapping information listed in Table 13, click on the button **Map a**

**Table 13.** Computation mapping, level L0

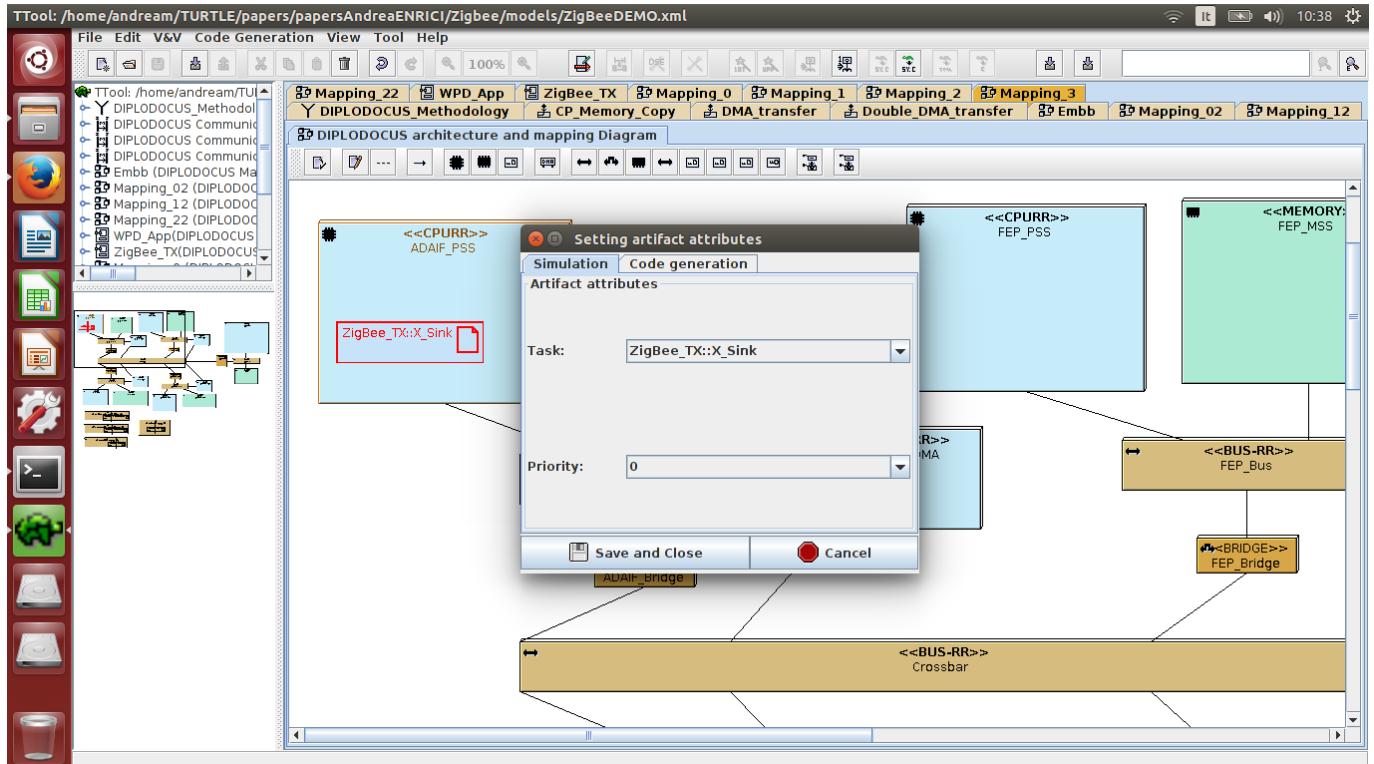
Task name	Task type	Architecture unit
X_Source	eXecution	Main CPU
F_Source	Firing	Main CPU
X_Symbol2ChipSeq	eXecution	Mapper PSS
F_Symbol2ChipSeq	Firing	Main CPU
X_Chip_to_Octet	eXecution	Interleaver PSS
F_Chip_to_Octet	Firing	Main CPU
X_CWL	eXecution	FEP PSS
F_CWL	Firing	Main CPU
X_CWP_I	eXecution	FEP PSS
F_CWP_I	Firing	Main CPU
X_CWL_Q	eXecution	FEP PSS
F_CWL_Q	Firing	Main CPU
X_Sink	eXecution	ADAIF PSS
F_Sink	Firing	Main CPU

task<sup>6</sup> (button n.7 in Fig. 43) and click on the platform unit where to map the computation. This attaches a UML artifact to the target platform unit. Double click on the artifact, and in the simulation tab select the task/primitive component that you want to map, then save and close. To demonstrate, Fig. 72 shows the mapping of X\_Txsink on the CPU unit that model the ADAIF computational core.

The memories used to store input/output data (mapping level L1) are chosen according to the access capabilities of each DSPU and of the main CPU. This results in a mapping where the local memory of each DSPU is used to store the input/output data for the computations that have been mapped onto the DSP's Processing SubSystem. For instance, task X\_Symbol2ChipSeq is mapped to the Mapper DSPU, therefore, the input/output data are mapped to the Mapper local Memory SubSystem (MSS). Table 14 summarizes this mapping. To implement the mapping of memory units, we must distinguish between two cases:

1. The mapped memory unit is directly connected (i.e., via one single bus unit) to the processing unit where the producer task will store its output data. In this case, we have a communication match and no Communication

<sup>6</sup> We specify here that this button maps a primitive component of the application model. The term *task* is synonymous for a primitive component of the application model. It is used in the tool for historical reasons as in the TML language, an application is described in terms of interconnected tasks rather than primitive components (UML/SysML terminology).



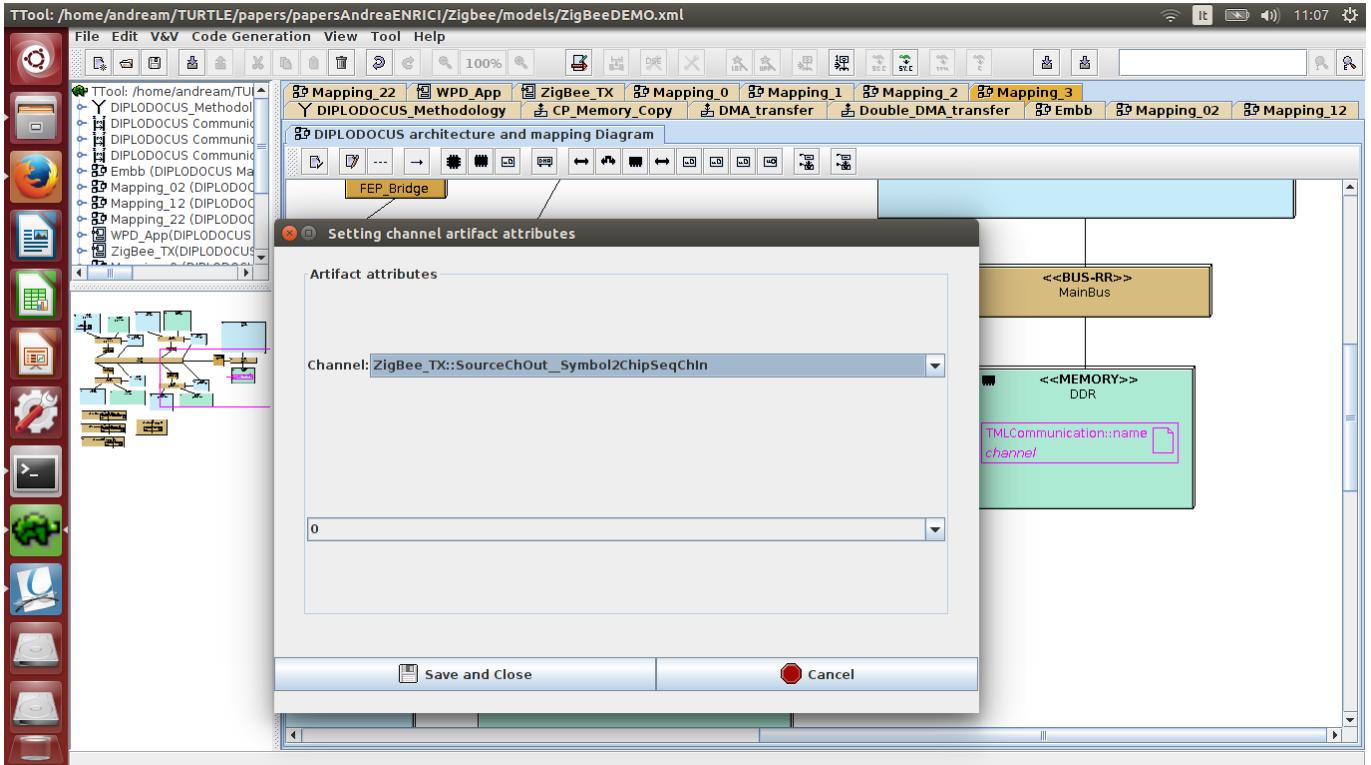
**Fig. 72.** The mapping of task TX\_Xsink on the ADAIF computational core

**Table 14.** Storage mapping, level L1

Task name	Input data memory	Output data memory
X_Source		Main memory
X_Symbol2ChipSeq	Mapper MSS	Mapper MSS
X_Chip_to_Octet	INTL MSS	INTL MSS
X_CWL	FEP MSS	FEP MSS
X_CWP_I	FEP MSS	FEP MSS
X_CWL_Q	FEP MSS	FEP MSS
X_Sink	ADAIF MSS	ADAIF MSS

Pattern is needed to model the transfer of data. We only need to map a data channel onto a target memory: click on button n.14 in Fig. 43, instantiate the mapping artifact for a data channel onto the target memory. Double click on the artifact and select the desired data channel, Fig. 73, then save and close.

- The mapped memory unit and the processing unit are not directly connected (i.e., the shortest path between the two units is composed of more than one bus). In this case, we are facing a communication mismatch and we must instantiate the block dedicated to map a Communication Pattern. Click on button n.12 in Fig. 43 and place the block in the mapping model. Then add onto this block the UML artifact to map a port, button n.15 in Fig. 43. Double click on the artifact and select the port<sup>7</sup> and the memory unit as shown in Fig. 74 for the mapping of port chip2octet\_ch\_in onto the local memory of the INTL DSP unit.



**Fig. 73.** The mapping of a memory unit in absence of a communication mismatch: mapping of a data channel

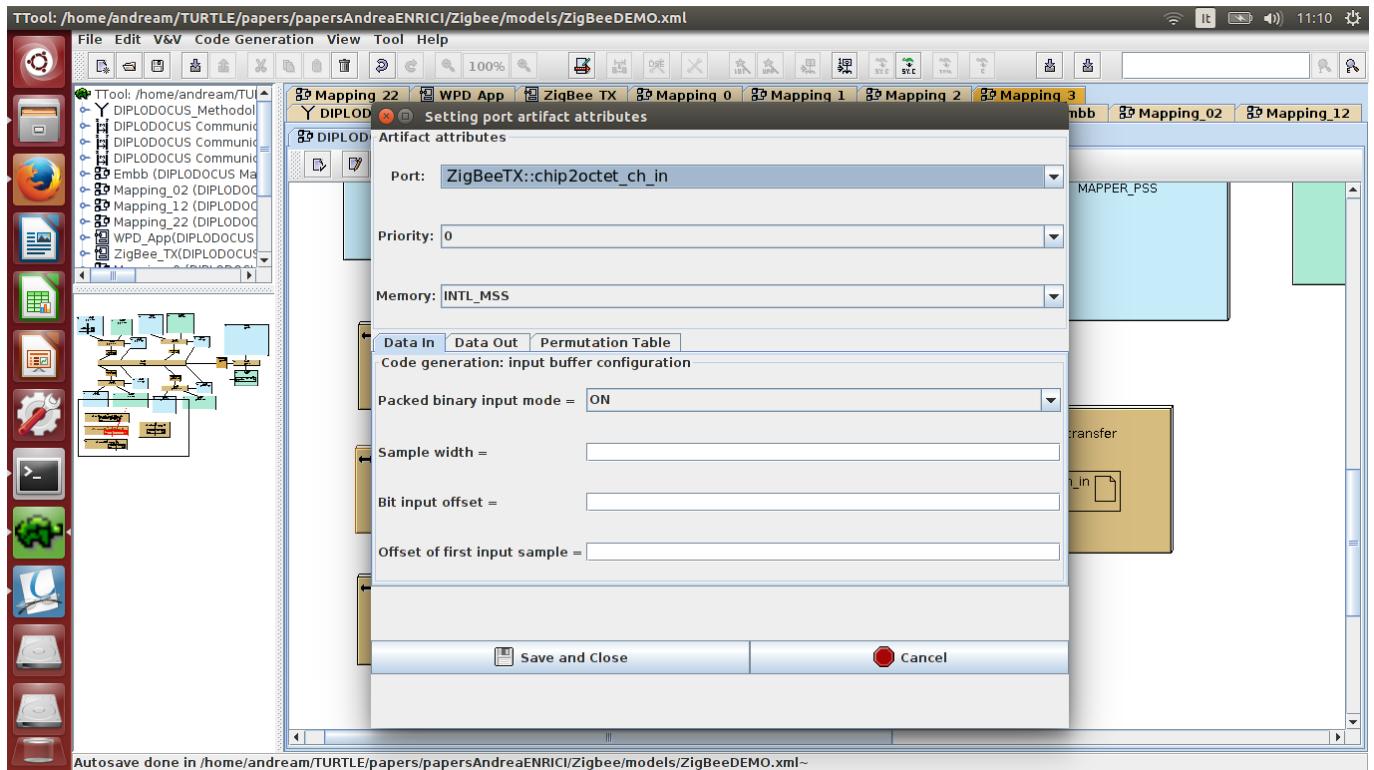
At this point, the mapping of the application model is completed and we can proceed to map the Communication Patterns from our library. For this purpose, we must first decide how data is transferred between DSPUs. Given the mapping of computations in table 13, EMBB's topology and the expertise of a platform programmer from our research group, we choose to use:

- The Main CPU to copy input samples to the Mapper DSPU
- The DMA engine of the Mapper to transfer chip sequences to the Interleaver
- The DMA engine of the Interleaver to transfer octets to the Front End Processor
- The DMA engine of the Front End Processor to transfer samples to the ADAIF

Therefore, from our library we instantiate and map 4 Communication Patterns:

- *CP01*: a memory copy CP that transfers the output data of task X\_Source. It is composed of: 1 controller instance (CPU\_Controller), 2 storage instances (Src\_Storage, Dst\_Storage) and 2 transfer instances.
- *CP02*: a DMA CP that transfers the output data of X\_Symbol2ChipSeq. It is composed of: 2 controller instances (CPU\_Controller, DMA\_Controller), 2 storage instances (Src\_Storage, Dst\_Storage) and 4 transfer instances.
- *CP03*: a DMA CP that transfers the output data of X\_Chip2Octet. It is composed of: 2 controller instances (CPU\_Controller, DMA\_Controller), 2 storage instances (Src\_Storage, Dst\_Storage) and 4 transfer instances.

<sup>7</sup> The tool lists the destination ports of each channel



**Fig. 74.** The mapping of a memory unit in presence of a communication mismatch: mapping a memory unit via the mapping block for Communication Patterns

- **CP04:** the sequence of two DMA CPs that transfer the output data of X\_CWP\_I and X\_CWP\_Q. It is composed of: 4 controller instances (2 CPU\_Controllers, 2 DMA\_Controller), 4 storage instances (2 Src\_Storage, 2 Dst\_Storage) and 8 transfer instances.

The above CPs are summarized also in Table 15 and Table 16. In the latter, the *transfer* instances of CP01 are those shown in the diagram in Fig. 57. The *transfer* instances of CP02 and CP03 are those shown in the Sequence Diagrams of Fig. 48, Fig. 49 and Fig. 50. We remind the reader (Property 17 in Appendix 1.A) that the name of an instance is unique only within a given Communication Pattern, not within a library of Communication Patterns, such as the one composed by CP01-CP04. Concerning CP04, *transfer* instances Transfer\_instance1-8 are those shown in Fig. 61-Fig. 66.

**Table 15.** List of the Communication Patterns that are part of the library of models for the case study

Identifier	Type	Mapped port	Description
<b>CP01</b>	Memory copy transfer as in Fig. 56	X_Symbol2ChipSeq input port	Transfer output data of X_Source
<b>CP02</b>	DMA transfer as in Fig. 47	X_Chip2Octet input port	Transfer output data of X_Symbol2ChipSeq
<b>CP03</b>	DMA transfer as in Fig. 47	X_CWL input port	Transfer output data of X_Chip2Octet
<b>CP04</b>	DMA transfer as in Fig. 58	X_Sink input port	Transfer output data of X_CWP_I and X_CWP_Q

We can now proceed to the Communication configuration mapping (level L3). Table 17 lists the values that are assigned to the attributes of the Communication Patterns. Given the mapping of computations at level L0, an exact value can be assigned to the amount of data to be transferred in a memory copy and a DMA transfer. Similarly, given the mapping of the storage instances at level L1, the source and destination addresses of a data transfer can be assigned. However, in Table 17 these addresses are left as unspecified because they are not taken into account by the

**Table 16.** List of the instances of the Communication Patterns in Table 15

Identifier	Controller instances	Storage instances	Transfer instances
<b>CP01</b>	CPU_Controller	Src_Storage_instance, Dst_Storage_instance	Transfer_instance1, Transfer_instance2
<b>CP02, CP03</b>	CPU_Controller, DMA_Controller	Src_storage_instance, Dst_storage_instance	Transfer_instance1, Transfer_instance2, Transfer_instance3, Transfer_instance4
<b>CP04</b>	CPU_Controller1, CPU_Controller2, DMA_Controller1, DMA_Controller2	Src_storage_instance1, Src_storage_instance2, Dst_storage_instance1, Dst_storage_instance2,	Transfer_instance1, Transfer_instance2, Transfer_instance3, Transfer_instance4, Transfer_instance5, Transfer_instance6, Transfer_instance7, Transfer_instance8

simulator engine in TTool/DIPLODOCUS. Instead, they are used as parameters for the automatic code generation as discussed in Section 9. In accordance with the abstraction level of a DIPLODOCUS application model, the simulator engine evaluates the impact of read/write transactions according to the amount of data to transfer, regardless of the specific memory location where data are read from or written to. The numerical value of variable `counter`<sup>8</sup> in Table 17

**Table 17.** List of the values assigned to the attributes of CP instances listed in Table 16, mapping level L2

Identifier	Attribute	Data to transfer	Value
<b>CP01</b>	<code>counter</code> <code>sourceAddress</code> <code>destinationAddress</code>	31 bytes	8 unspecified unspecified
<b>CP02</b>	<code>counter</code> <code>sourceAddress</code> <code>destinationAddress</code>	31 64-bit data words	1 unspecified unspecified
<b>CP03</b>	<code>counter</code> <code>sourceAddress</code> <code>destinationAddress</code>	248 64-bit data words	1 unspecified unspecified
<b>CP04</b>	<code>counter1</code> , <code>counter2</code> <code>sourceAddress1</code> <code>sourceAddress2</code> <code>destinationAddress1</code> <code>destinationAddress2</code>	2048 32-bit samples 1922 32-bit samples	2 2 unspecified unspecified unspecified unspecified

is explained by the following analysis for the transfer of a packet with a 25-bytes payload and a 6-bytes header. The main CPU must send 25 bytes to the MAPPER MSS. Being a 32-bit CPU, this unit performs 8 memory accesses (value of `counter` for CP01 in Table 17).

The MAPPER first transforms the received 31 bytes into 62 4-bit symbols and then further transforms each symbol into a 32 bit data word (i.e., a chip sequence). Therefore, the MAPPER must send 62 32-bit data words to the INTL MSS. Given the crossbar width of 64 bits, this results into one DMA transfer<sup>9</sup> of 31 64-bit data words (value of `counter` for CP02 in Table 17).

The INTL transforms each received bit into an octet and sends these data to the FEP MSS over the 64-bit crossbar.

<sup>8</sup> We remind the reader that this variable defines the number of DMA transfer cycles, as referenced by diagrams DMACycle\_SD1 and DMACycle\_SD2 in Fig. 58

<sup>9</sup> Each DMA unit in EMBB can transfer up to 1024 bytes in one DMA cycle.

For the initial message of 31 bytes (i.e., 25 bytes payload and 6 bytes header) this results into a DMA transfer of 248 64-bit data words (value of `counter` for CP03 in Table 17).

Finally, the FEP substitutes each received octet by a set of 4 16-bit integers (the half-sine is represented as a set of 4 samples). These integers are intertwined after the pre-computed header, with a shift equal to two samples in order to realize the delay between the I and Q branches and compose blocks of 32-bit complex data (i.e., 16 bits for the I branch and 16 bits for the Q branch). Overall, considering the header and the I/Q delay, this amounts to 3970 32-bit complex samples (sum of the data to transfer, `counter1` and `counter2`, for CP04 in Table 17). The FEP unit assembles these samples in chunks composed of 1024 samples and dispatches them to the ADAIF over the 64-bit crossbar via a sequence. To conclude, this results in 3 1024-words DMA transfers and one 946-words DMA transfer, amounting to 4 DMA transfers (values of `counter1` and `counter2` for CP04 in Table 17).

Table 18 lists the mapping needed to configure the communications. CP04 constitutes a special case as it models a double DMA data-transfer where instead of only one, there are two CPU controller instances, two DMA controller instances, two source storage instances and two destination storage instances. However, each pair of such instances is mapped onto the same platform unit: for both DMA transfers the DMA controller of the FEP unit is used. These DMA transfers are configured by the Main CPU and transfer data from the FEP MSS to the ADAIF MSS. The transfers are executed sequentially as a consequence of the fact that the FEP unit is capable of executing one processing operation at a time. To map the instances as listed in Table 18, we must use the mapping blocks for Communication Patterns

**Table 18.** Communication configuration mapping for CPs, level L2

Identifier	CPU Controller instance(s)	DMA Controller instance(s)	Source Storage instance(s)	Destination Storage instance(s)
<b>CP01</b>	Main CPU		Main Memory	MAPPER MSS
<b>CP02</b>	Main CPU	MAPPER DMA	MAPPER MSS	INTL MSS
<b>CP03</b>	Main CPU	INTL DMA	INTL MSS	FEP MSS
<b>CP04</b>	Main CPU	FEP DMA	FEP MSS	ADAIF MSS

that we instantiated before when mapping the memory units. Double click on the block (not on the port mapping artifact) to open the mapping interface then select the CP, the corresponding instances and attributes to map. At

**Table 19.** Routing mapping for CPs, level L3

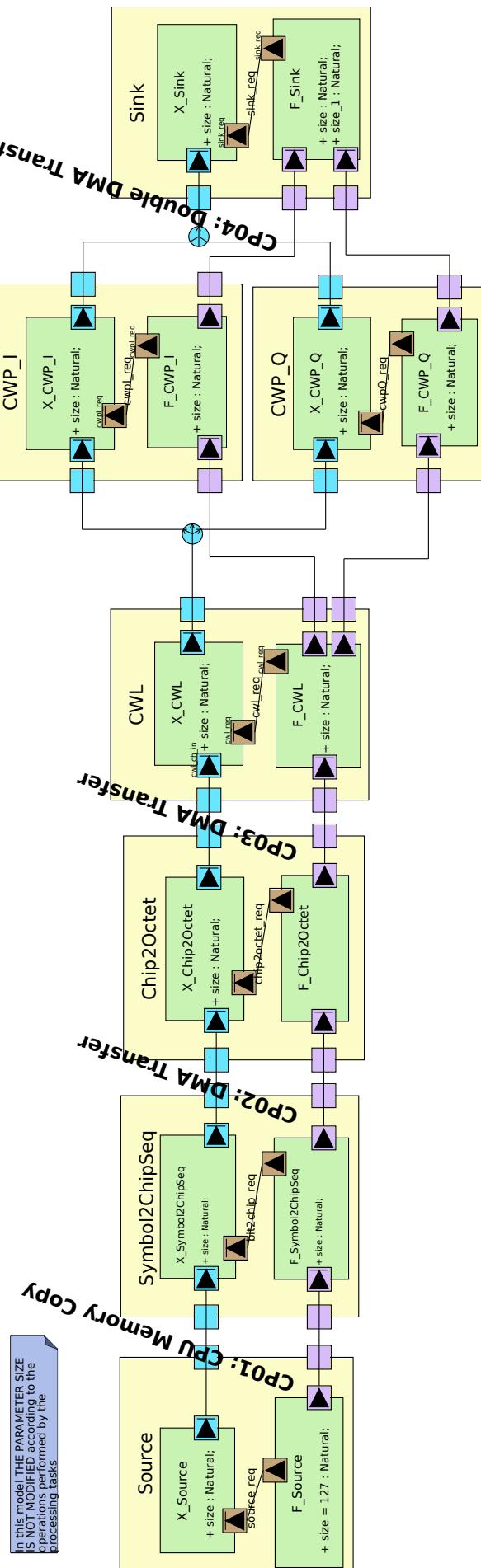
Identifier	Transfer instance 1	Transfer instance 2	Transfer instance 3	Transfer instance 4
<b>CP01</b>	Main Bus Main Bridge Crossbar Mapper Bridge Mapper Bus	Main Bus Main Bridge Crossbar MAPPER Bridge MAPPER Bus		
<b>CP02</b>	Main Bus Main Bridge Crossbar MAPPER Bridge MAPPER Bus	MAPPER Bus	INTL Bus INTL Bridge Crossbar MAPPER Bridge MAPPER Bus	MAPPER Bus MAPPER Bridge Crossbar Main Bridge Main Bus
<b>CP03</b>	Main Bus Main Bridge Crossbar INTL Bridge INTL Bus	INTL Bus	FEP Bus FEP Bridge Crossbar INTL Bridge INTL Bus	INTL Bus INTL Bridge Crossbar Main Bridge Main Bus
<b>CP04</b>	Main Bus Main Bridge Crossbar FEP Bridge FEP Bus	FEP Bus	FEP Bus FEP Bridge Crossbar ADAIF Bridge ADAIF Bus	FEP Bus FEP Bridge Crossbar Main Bridge Main Bus

this point, we can perform the last mapping step, level L3. This mapping is summarized in Table 19 that shows the binding of transfer instances to platform units.

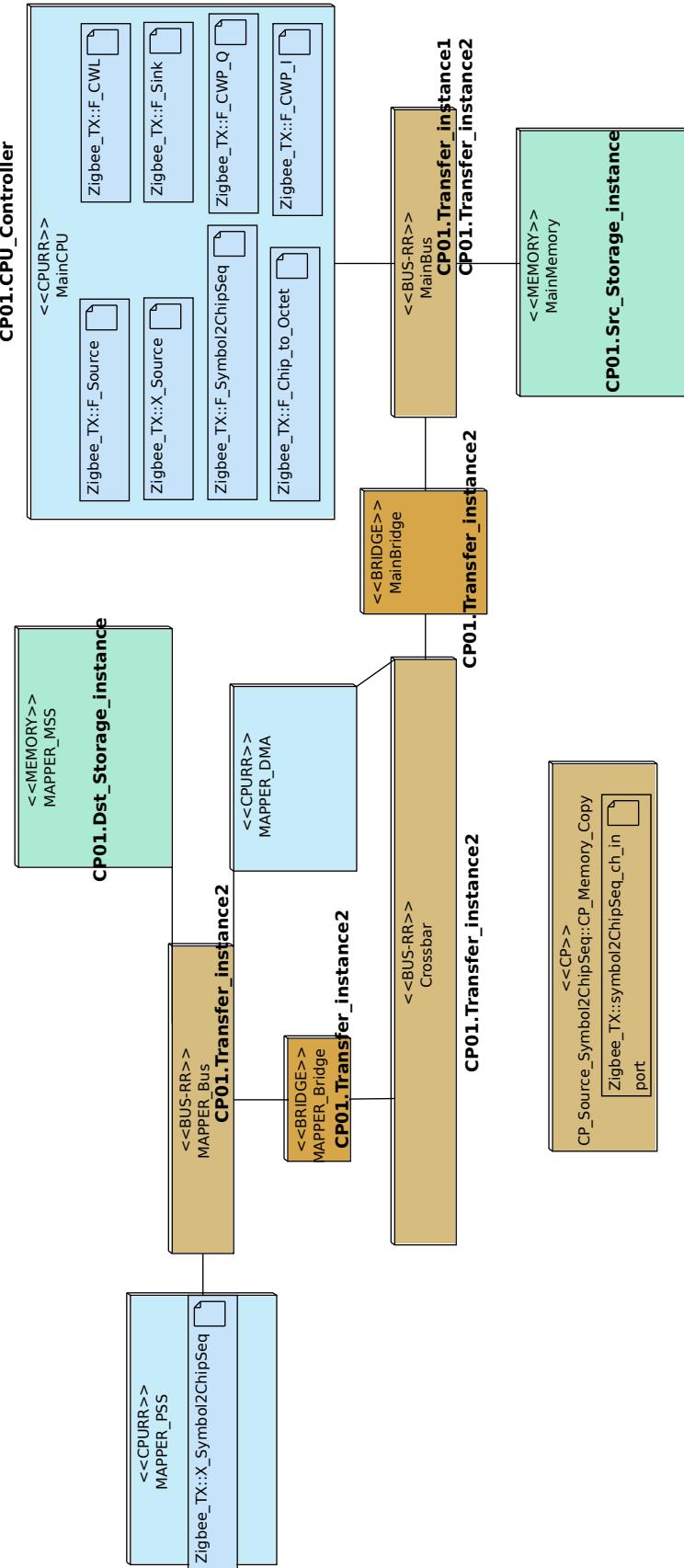
In Table 19 the mapping information of CP04 has been compacted so as to fit the information for the other Communication Patterns, although CP04 models a double DMA transfer. Such a transfer is in fact the sequence of two identical transfers. Therefore the instances of these two transfers are mapped to the same units and the attributes are assigned the same values. The mapping information for Transfer\_instance5, Transfer\_instance6, Transfer\_instance7 and Transfer\_instance8 is not displayed in Table 19 but is the same as, respectively, the mapping for Transfer\_instance1, Transfer\_instance2, Transfer\_instance3 and Transfer\_instance4.

Similarly to the mapping at level L2, to map the remaining instances as listed in Table 19, use the mapping blocks for Communication Patterns.

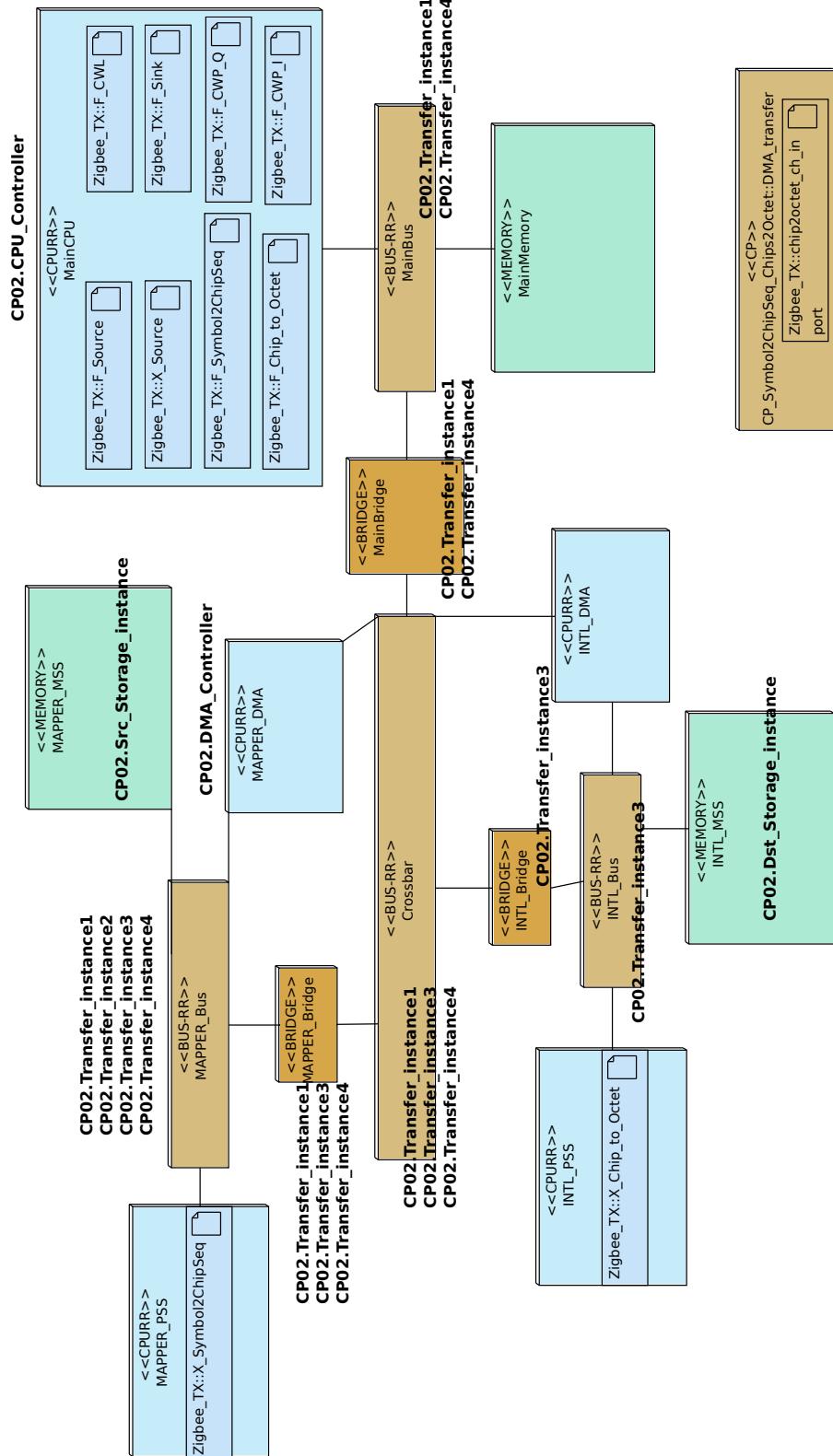
To visually help the reader to understand the mapping of Communication Patterns, Fig. 75 shows the application model with the CPs mapped to the data-channels. Fig. 76 to Fig. 79 each show the instances of CP01-CP04 mapped onto the platform model.



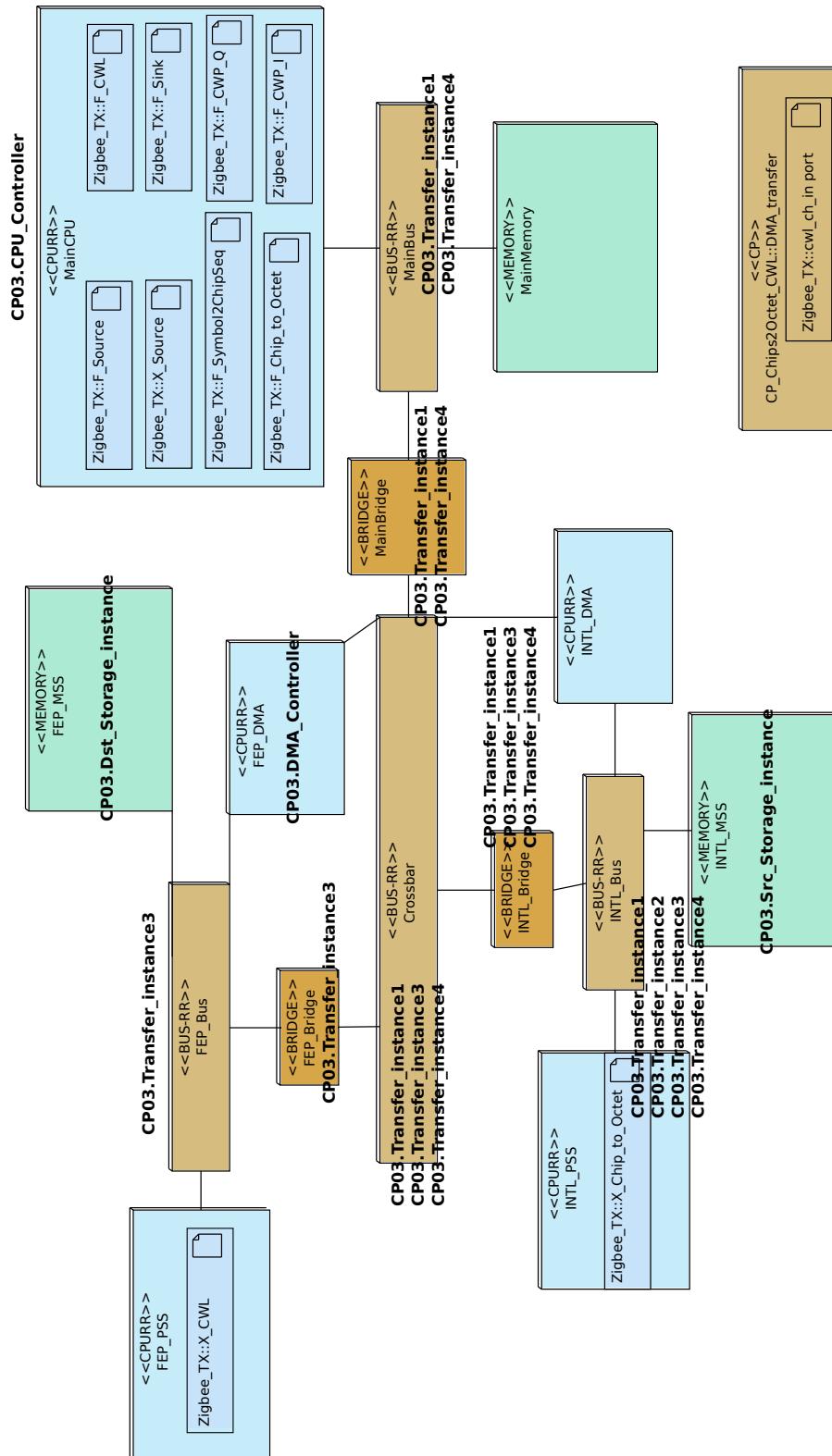
**Fig. 75.** The application model with CPs mapped to the data-channels for which a communication mismatch is encountered at mapping step.



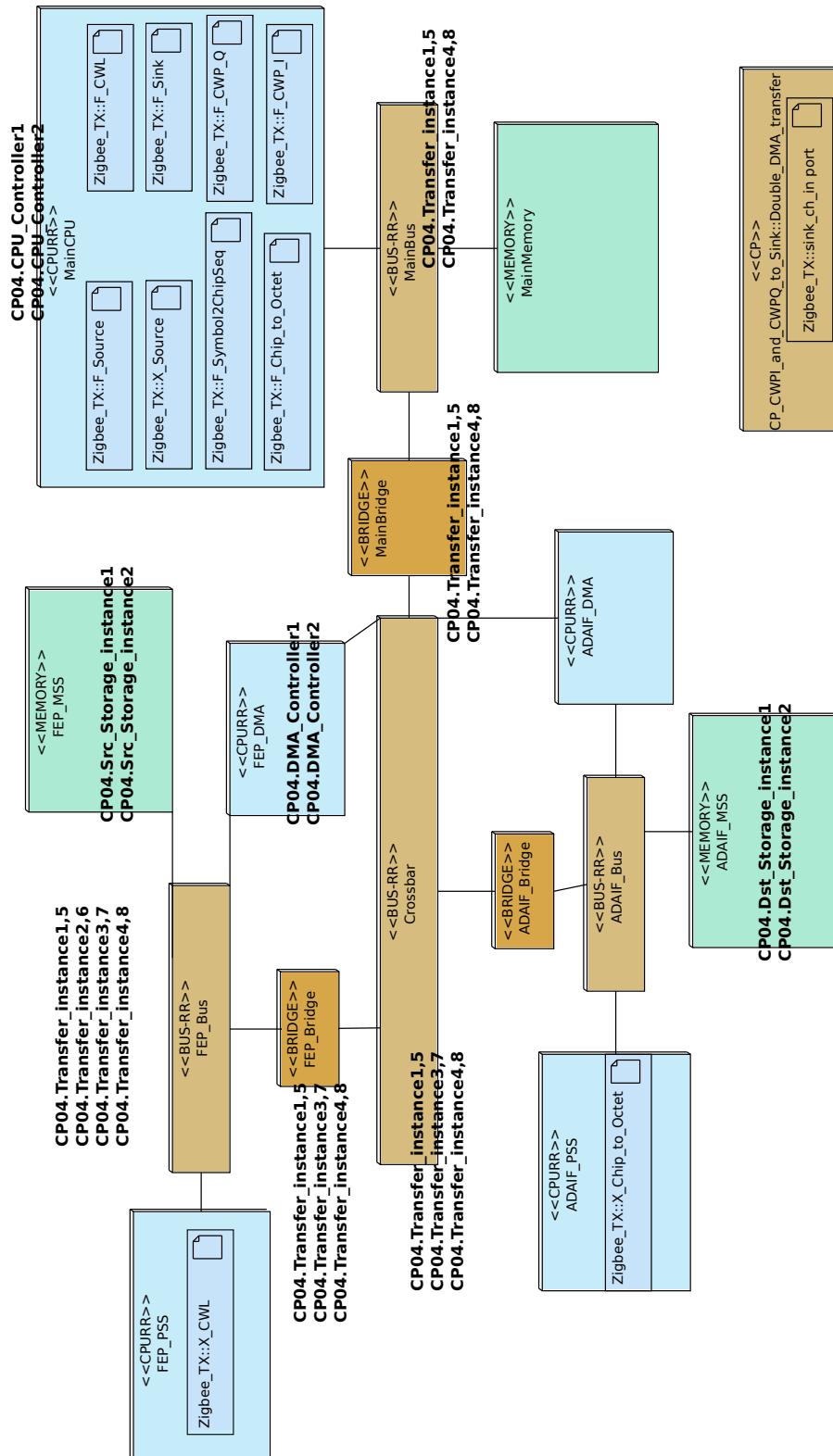
**Fig. 76.** An excerpt of EMBB's platform model shwoing the complete mapping of CP01's instances. CP01 is deployed to transfer data from X\_Source task (producer) to X\_Symbol2ChipSeq task (consumer).



**Fig. 77.** An excerpt of EMBB's platform model showing the complete mapping of CP02's instances. CP02 is deployed to transfer data from X\_Symbol2ChipSeq task (producer) to X\_Chip\_to\_Octet task (consumer).



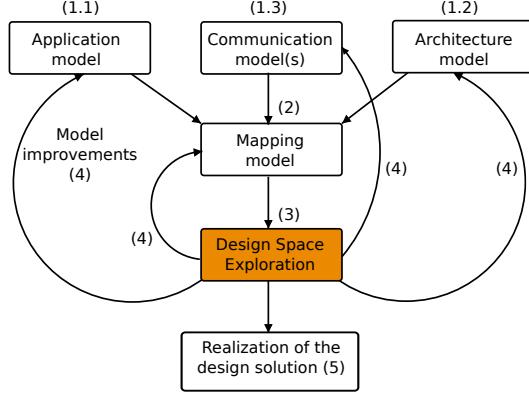
**Fig. 78.** An excerpt of EMBB's platform model showing the complete mapping of CP03's instances. CP03 is deployed to transfer data from X\_Chip\_to\_Octet task (producer) to X\_CWL task (consumer).



**Fig. 79.** An excerpt of EMBB's platform model showing the complete mapping of CP04's instances. CP04 is deployed to transfer data from X\_CWP\_I and X\_CWP\_Q tasks (producers) to X\_Sink task (consumer).

## 8 Design Space Exploration in TTool/DIPLODOCUS

Design Space Exploration (DSE), Fig. 80, is the activity of exploring design alternatives prior to implementation. In TTool/DIPLODOCUS this exploration is conducted via simulation, sub-section 8.1 and formal verification, sub-section 8.2.



**Fig. 80.** The step of Design Space Exploration, that is described in this section, in the context of the  $\Psi$ -chart design approach

### 8.1 Simulation

The simulation environment of TTool/DIPLODOCUS, allows an interactive exploration of an application after and before mapping onto a particular platform. The simulator engine [3] is based on *transactions*, a data structure that represents a computation or communication action involving one or more hardware components from a target platform. The use of this particular MoC results into the simulation speed directly matching the abstraction level of models. The simulator tool comes with a Graphical User Interface which allows the animation of application and platform models whose execution can be customized by means of breakpoints, generation of execution traces, save/restore of simulation states and other debug facilities. Simulation results are propagated back to the original UML/SysML models, so that the user does not have to be aware of the details internal to the simulator's executable models. This interactive simulation is accomplished at run-time, when models are animated to illustrate simulation progress.

The functional simulator of TTool/DIPLODOCUS is similar to the clock-cycle accurate simulation semantics of SystemC [19], however, it is specifically tailored for the semantics of high-level models of DIPLODOCUS [3]. As a matter of fact, it renounces many features of the standard SystemC simulation kernel (e.g., sensitivity to signals, repeated execution of threads until a steady state is reached, management of different concurrency primitives, explicit representation of events, event queues, creation and cancellation of events, truncation of transactions) in order to improve simulation speed and reduce the number of simulation threads that would normally be associated to active units such as CPUs in the architecture model.

Appendix 1.B reports an excerpt from [4] that details the semantics of the simulation engine. We specify here that simulation outputs a trace that represents a single execution of the system under design.

Before launching the simulator, we must check the syntax of our design (e.g., diagrams are well connected). For this purpose, from the mapping diagram, press the dedicated button shown in Fig. 81. This opens up a window, Fig. 81, that allows a user to select the diagrams to check. By default all diagrams that are open for a given project are included, so let's click on the **Start Syntax Analysis** button. If you have followed the instructions that have been given so far, there should be no syntax error. In case the syntax check terminates with errors, these will be listed in the project navigation window under the section **Syntax analysis**. By double clicking on each error in the list, the tool will automatically redirect the user to the part of the design where the error resides, highlighting the error in red, whenever possible.

It is also possible to launch the syntax analysis from the single application, platform or Communication Pattern

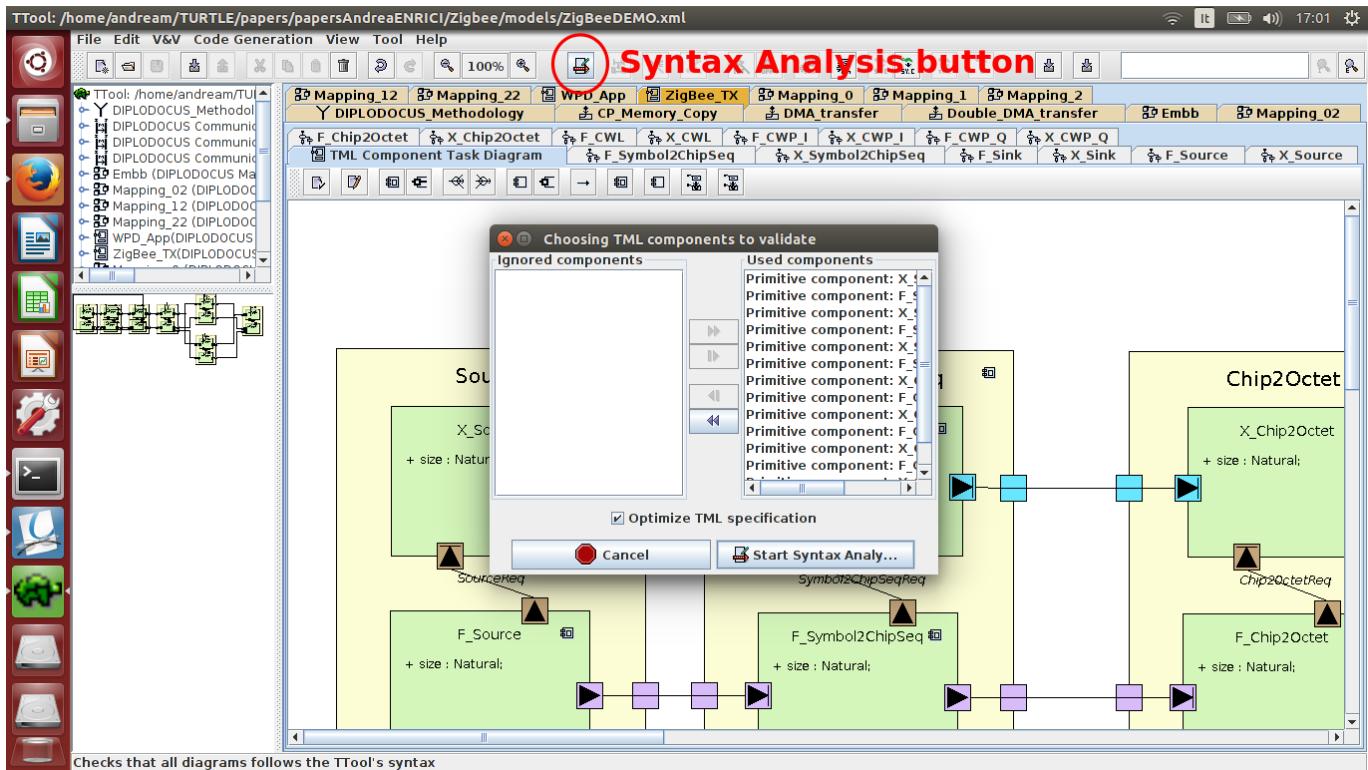


Fig. 81. Checking the syntax of the ZigBee design.

panels. In this case, these panels will be checked against syntax errors individually. Instead, syntax analysis from the mapping model checks the validity of the whole design (platform model and all the references to the mapped diagrams).

After syntax analysis is performed, models must be translated into input code for the simulator. This is done by clicking on the button **Generate code for simulation** that opens the window displayed in Fig. 82. In tab **Generate code**, you can select the location where code will be generated, the simulator version and to remove old compilation files. Normally, the user should use the default parameters that are taken from the configuration file. Clicking on the **Start** button launches the code generation process. Once completed, the tool automatically switches to tab **Compile** that allows the user to chose the location of the files to be compiled and the compilation command<sup>10</sup>. Again, clicking on the **Start** button launches the compilation process and once it has completed, the tool automatically switches to the last tab, **Execute**. The latter allows to select options to run the simulator Graphical User Interface. Unless the user has changed some parameters in the previous steps, he/she should use the default options. Click on the **Start** button to launch the simulator interface and then click on the button **Connect to simulator**. The simulator GUI is shown in Fig. 83. Before commenting about the simulation results of the ZigBee transmitter, we detail the simulator's GUI. We first describe the **Commands** panel and then the **Simulation information** panel.

Fig. 84 shows the **Control** tab of panel **Commands** and enumerates the available buttons:

1. Reset simulation
2. Stop simulation
3. Run simulation
4. Run to time x
5. Run x time units
6. Run x transactions
7. Run x commands
8. Run exploration
9. Run until CPU executes

<sup>10</sup> The default compilation command calls a Makefile that has been tested in Linux, MacOS and Cygwin for Windows. If you have a different configuration (e.g., Windows 10 without cygwin), then you need to adapt the compilation command. Also, we strongly recommend to use *gcc* (and not e.g. *clang*)

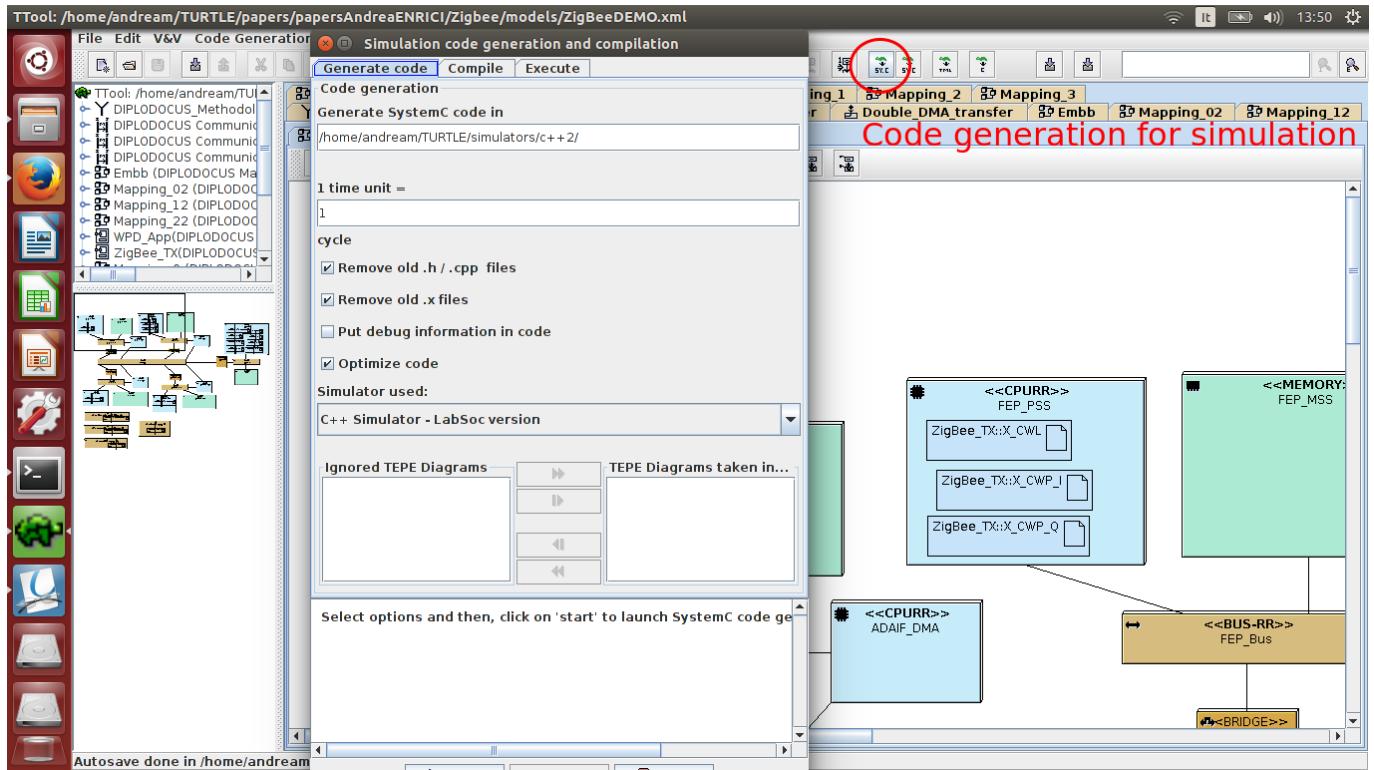


Fig. 82. Generating the code for simulation

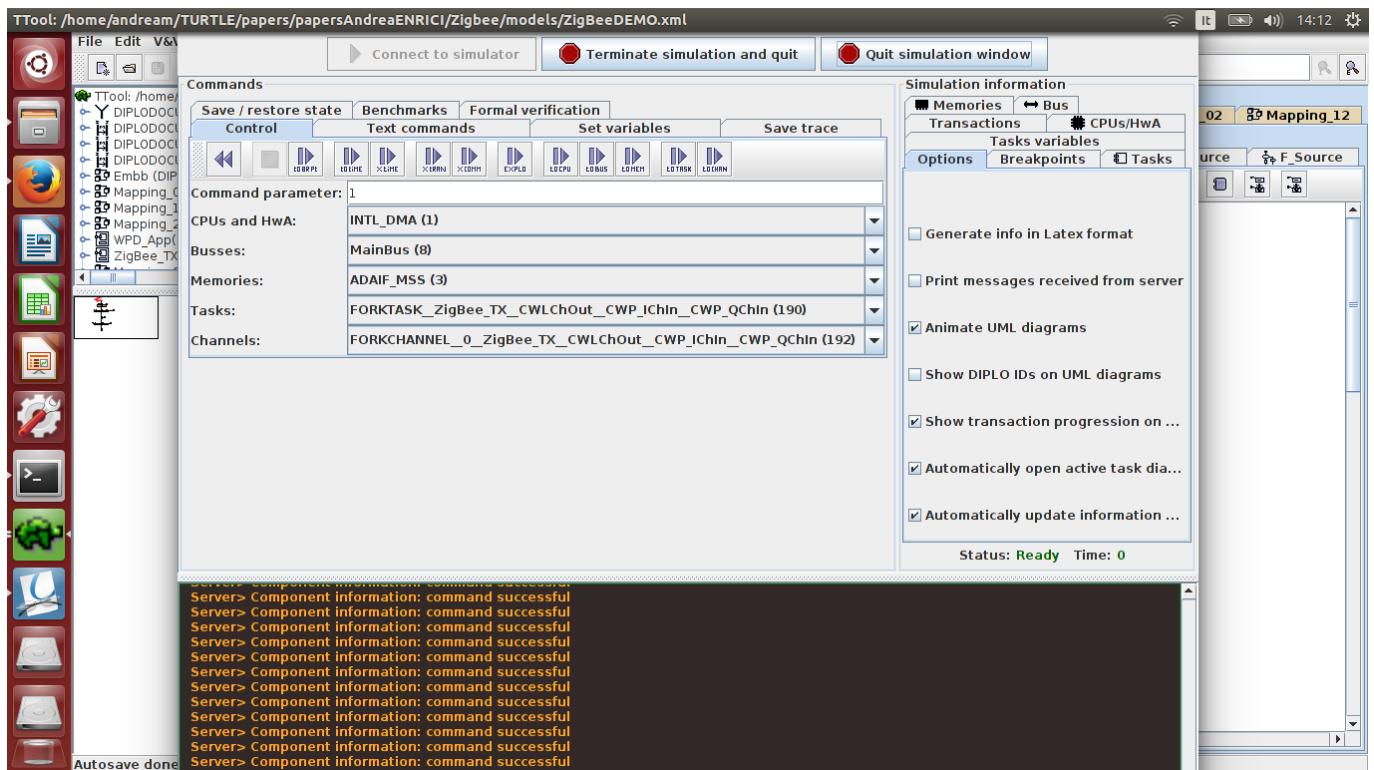
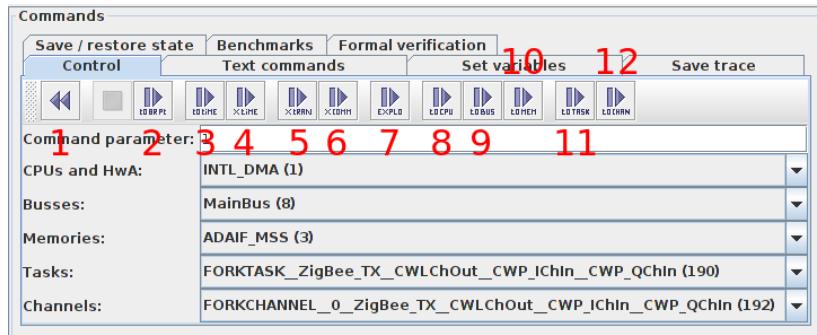


Fig. 83. The simulator Graphical User Interface

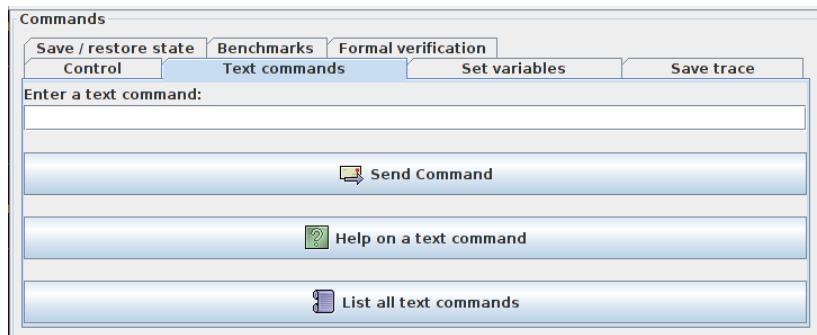
10. Run until transfer on bus
  11. Run until a memory access is performed
  12. Run until a task executes
  13. Run until a channel is accessed

For buttons 4, 5, 6 and 7, the term  $x$  is the number in the box **Command parameter**. For buttons 9, 10, 11, 12 and 13, the CPU, bus, memory, task and channel are those that can be selected below the buttons, respectively. Fig. 85 shows

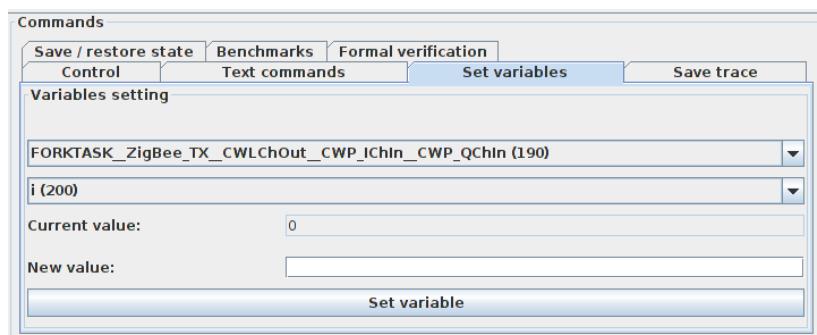


**Fig. 84.** The control tab in the simulator Graphical User Interface

the Text commands tab of panel Commands. Fig. 86 shows the Set variables tab of panel Commands. Fig. 87 shows



**Fig. 85.** The text commands tab in the simulator Graphical User Interface

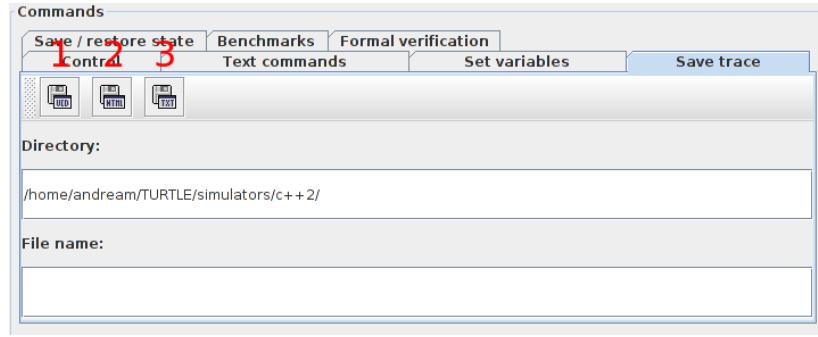


**Fig. 86.** The set variables tab in the simulator Graphical User Interface

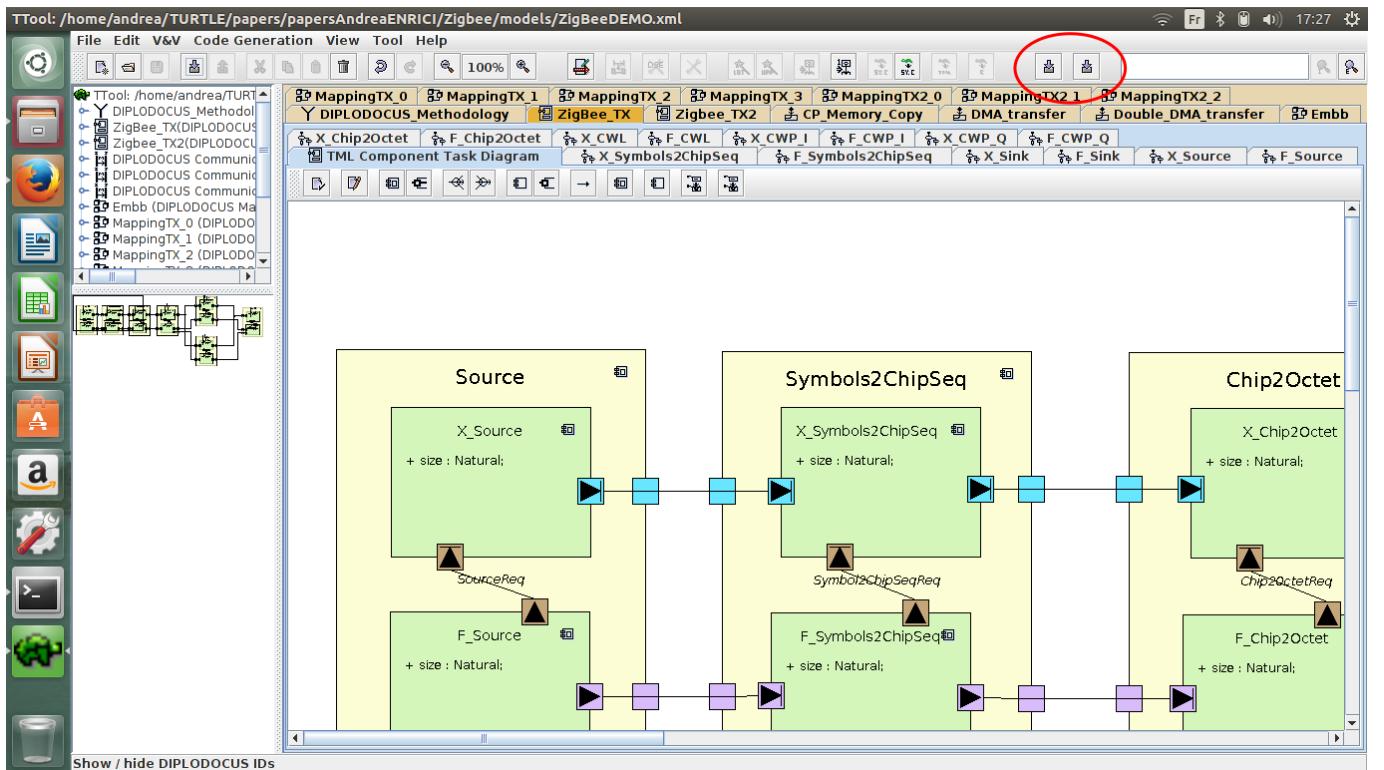
the **Save trace** tab of panel **Commands**. The buttons, listed below, allow a user to select the format, location and name

of the simulation trace to be saved. The VCD format (Value Change Dump) is typical of Verilog and VHDL tools and saves the trace in a waveform format that can be visualized with external tools such as GTKWave [20]. The latter can also be called from TTool. Two buttons exist, Fig. 88 that the user can configure to launch custom commands from the tool's GUI.

1. Save trace in VCD format
2. Save trace in HTML format
3. Save trace in TXT format



**Fig. 87.** The save trace tab in the simulator Graphical User Interface

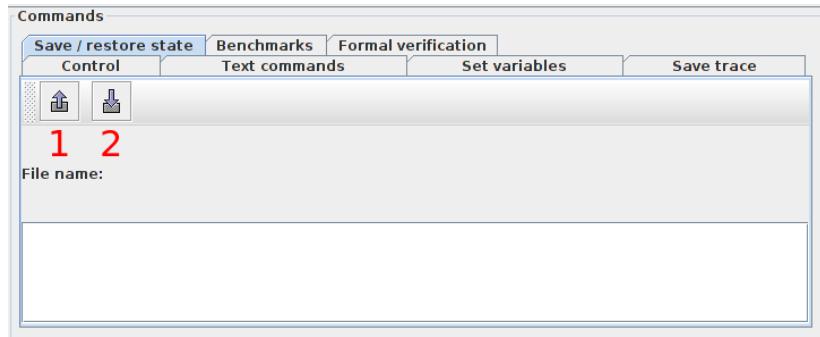


**Fig. 88.** The buttons used to launch custom commands inside TTool/DIPLODOCUS

Fig. 89 shows the **Save/restore state** tab of panel **Commands**. The buttons, listed below, allow the user to save and re-load a simulation trace from a given location (complete path name of the file).

1. Save simulation state in file

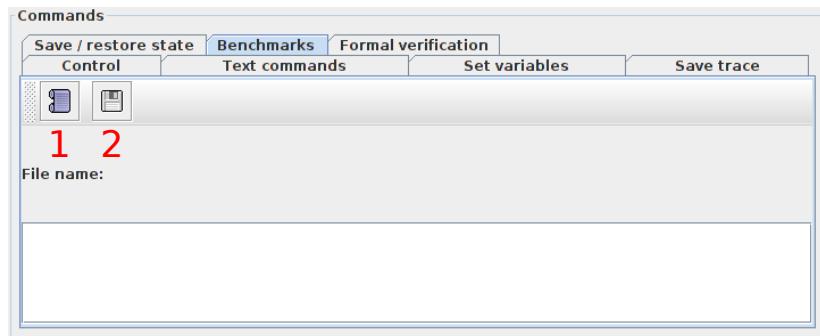
2. Restore simulation state from file



**Fig. 89.** The Save/Restore state tab in the simulator Graphical User Interface

Fig. 90 shows the **Benchmarks** tab of panel **Commands**. The buttons, listed below, allow a user to print and to save a benchmark file.

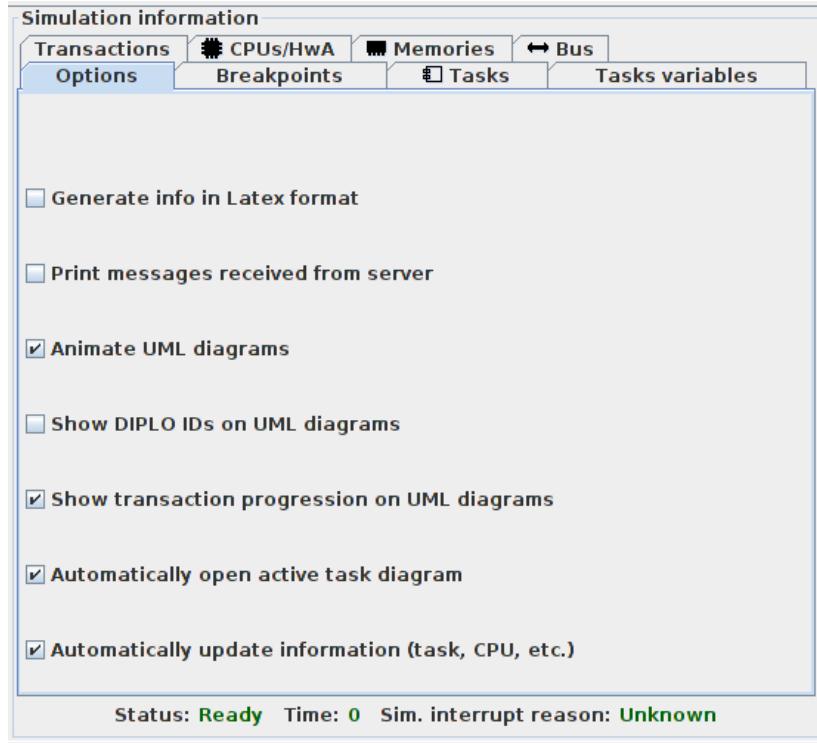
1. Print benchmark
2. Save benchmark



**Fig. 90.** The benchmarks tab in the simulator Graphical User Interface

The tab dedicated to **Formal verification** is described in subsection 8.2.

Fig. 91 shows the Options tab of panel **Simulation information**. Fig. 92 shows the Breakpoints tab of panel



**Fig. 91.** The options tab in the simulator Graphical User Interface

**Simulation information.** Breakpoints can be added to operators of the application diagram (e.g., write to channel, receive event) before generating the code for simulation by right-clicking on the operator and selecting **Add/remove breakpoint**. They can also be set during simulation using the same graphical way. Alternatively, they can also be added during simulation by selecting the task/primitive component, then the desired operator and by clicking on the button **Add** in Fig. 92. Fig. 93 shows the Tasks tab of panel **Simulation information**. These tab lists all the tasks/primitive components of the design being simulated, as well as their ID, state and the number of cycles executed onto the mapped CPU unit. Fig. 94 shows the Task variables tab of panel **Simulation information**. This tab allows us to monitor the attributes of each task/primitive component. The tab lists the task names and IDs as well as the attributes (variables) names, IDs and values. Fig. 95 shows the Transactions tab of panel **Simulation information**. This tab lists all the transactions that have been executed by the at a given moment in simulation time. The list shows the task, the mapped platform unit, the command that triggered the transaction, the start and end times, the length and the channel involved in the transaction. Fig. 96 shows the CPUsHwA tab of panel **Simulation information**. This tab shows the internal status of each CPU and HwA unit of the platform model.

Fig. 97 shows the **Memories** tab of panel **Simulation information**. This tab shows the internal status of each memory unit of the platform model that has been accessed at a given moment in simulation time.

**The simulation results of the ZigBee transmitter (physical layer)** The simulation results of the mapping model described in Section 7 are summarized in Table 20. As expected, the FEP\_PSS is the most charged unit of the whole system as it is assigned the most computationally heavy tasks. Although all the control tasks (Firing tasks) are mapped onto the MainCPU, the load of this unit is comparable to that of the ADAIF\_PSS and the MAPPER\_PSS. In terms of communications, the Crossbar is the most charged bus unit as it is the intersection of the data traffic of the whole platform. The load of the other bus units is due to accesses to the DSPU local memory units. Instead, the load of the DMA units ranges from 3% to 7%.

Fig. 98 shows an excerpt of the simulation waveform that displays the activity of the Crossbar, the DMA units, the Main CPU and the Main Bus. It can be seen the correspondence between activity in the Crossbar and the Main Bus units and activity in the DMA units and the Main CPU corresponding to the data transfers modeled by the

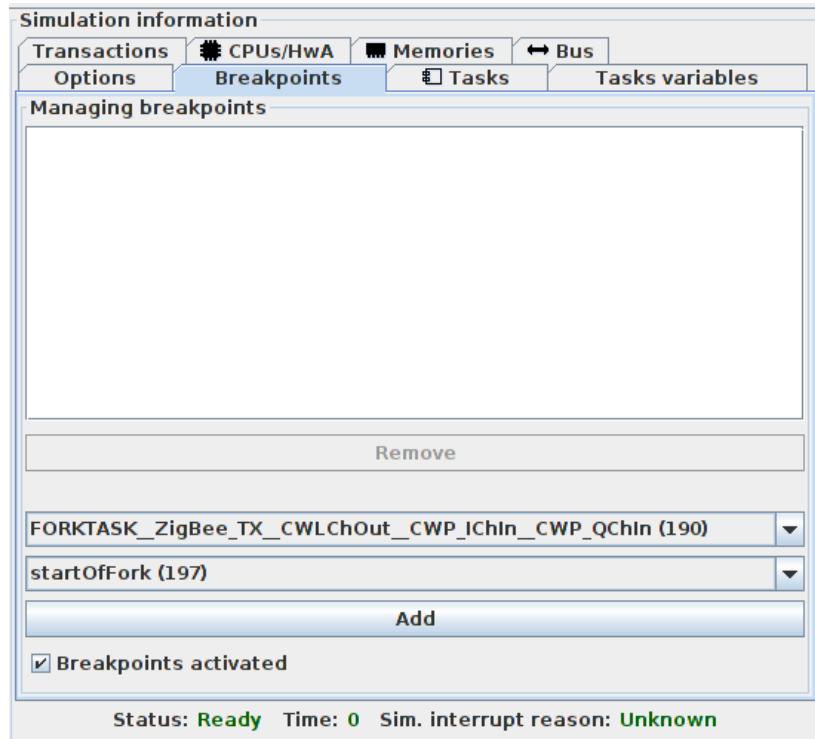


Fig. 92. The breakpoints tab in the simulator Graphical User Interface

Simulation information				
Transactions	CPUs/HwA	Memories	Bus	
Options	Breakpoints	Tasks	Tasks variables	
				Executed cycles
Task Name	Task ID	State		
FORKTASK_Zi...	190	terminated	0	
JOINTASK_Zi...	204	terminated	0	
ZigBee_TX_F...	38	terminated	0	
ZigBee_TX_F...	32	terminated	0	
ZigBee_TX_F...	29	terminated	0	
ZigBee_TX_F...	35	terminated	0	
ZigBee_TX_F...	47	terminated	0	
ZigBee_TX_F...	44	terminated	0	
ZigBee_TX_F...	50	terminated	0	
ZigBee_TX_X...	56	terminated	0	
ZigBee_TX_X...	62	terminated	0	
ZigBee_TX_X...	59	terminated	0	
ZigBee_TX_X...	53	terminated	0	
ZigBee_TX_X...	26	terminated	0	
ZigBee_TX_X...	41	terminated	0	
ZigBee_TX_X...	65	terminated	0	

Update task information

Status: Ready Time: 0 Sim. interrupt reason: Unknown

Fig. 93. The tasks tab in the simulator Graphical User Interface

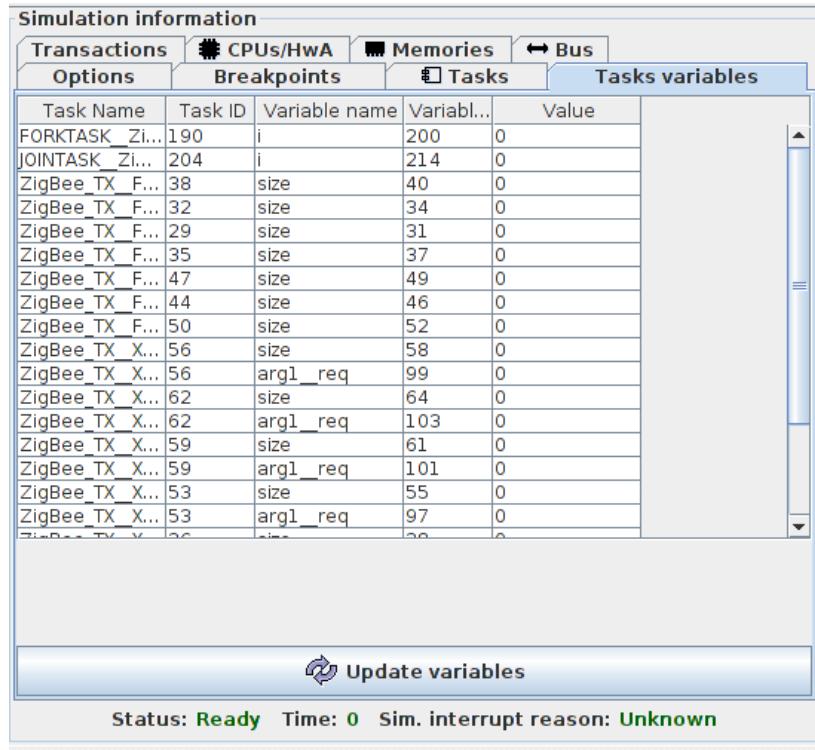


Fig. 94. The task variables tab in the simulator Graphical User Interface

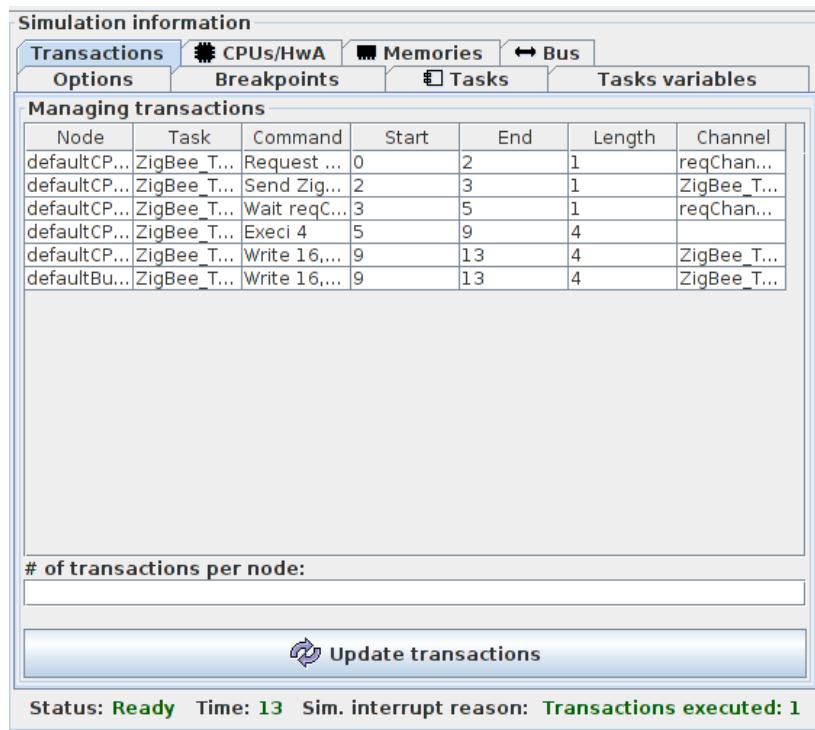
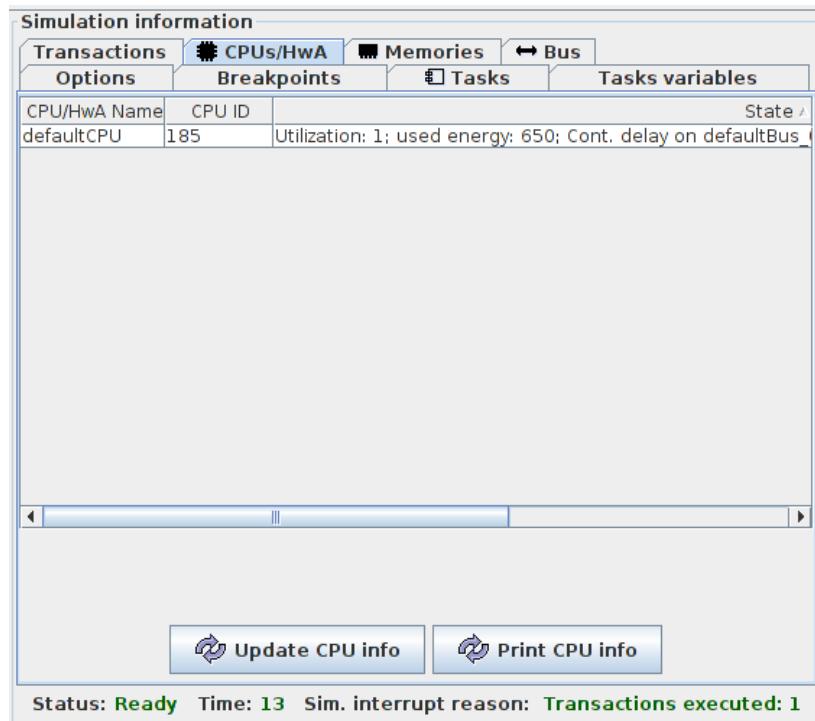
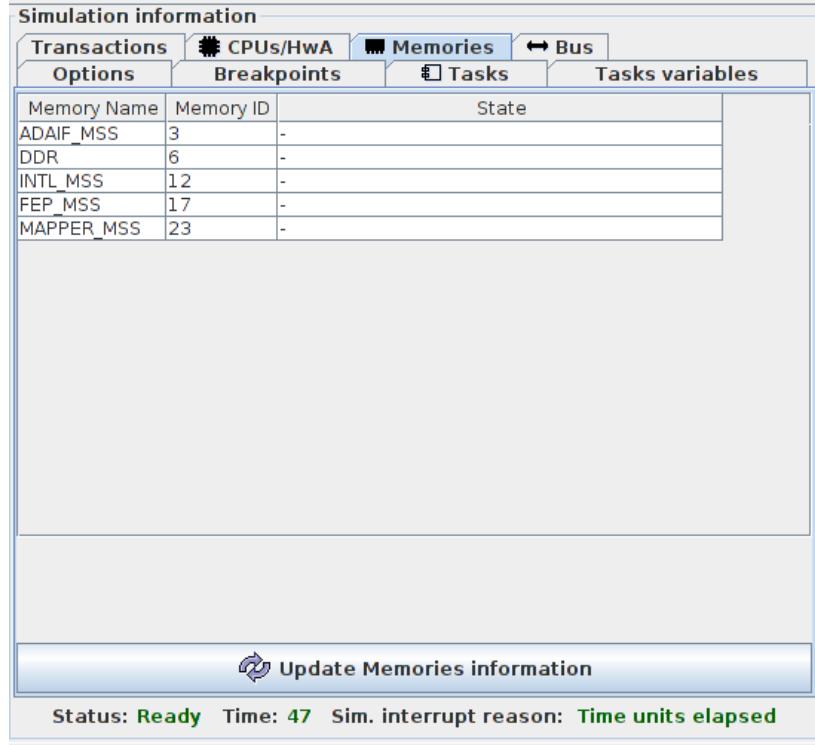


Fig. 95. The transactions state tab in the simulator Graphical User Interface



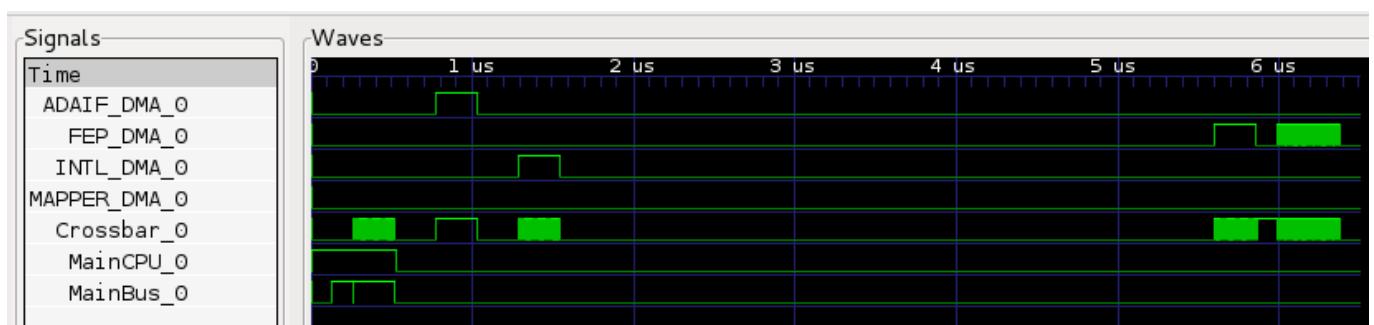
**Fig. 96.** The CPUs/HwA tab in the simulator Graphical User Interface



**Fig. 97.** The memories tab in the simulator Graphical User Interface

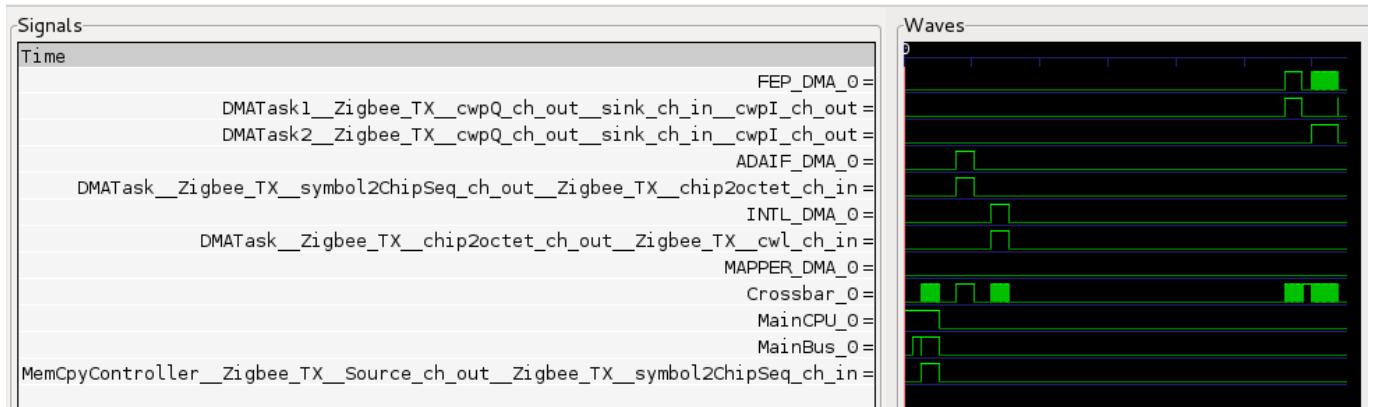
**Table 20.** The simulation results for the mapping configuration described in Section 7

Architecture unit	Load
Crossbar	13%
ADAIF_PSS	3%
ADAIF_Bus	9%
ADAIF_DMA	0%
INTL_PSS	5%
INTL_Bus	7%
INTL_DMA	3%
MAPPER_PSS	5%
MAPPER_Bus	7%
MAPPER_DMA	3%
FEP_PSS	67%
FEP_Bus	15%
FEP_DMA	7%
MainCPU	7%
MainBus	5%

**Fig. 98.** An excerpt of the simulation waveform showing the activity of the platform units for the design of the ZigBee transmitter, i.e., data transferred by the Crossbar, the DMA units, the Main Bus and the Main CPU.

Communication Patterns.

Fig. 99 instead shows the correspondence between activity in the units of the architecture model and the TML tasks



**Fig. 99.** An excerpt of the simulation waveform showing the correspondence between activity of the architecture units and the TML tasks equivalent to a Communication Pattern.

corresponding to the Communication Patterns<sup>11</sup>.

In the next subsection, we will discuss how the above simulation results can be used to drive the modification of models, with a focus on Communication Patterns.

## 8.2 Formal verification

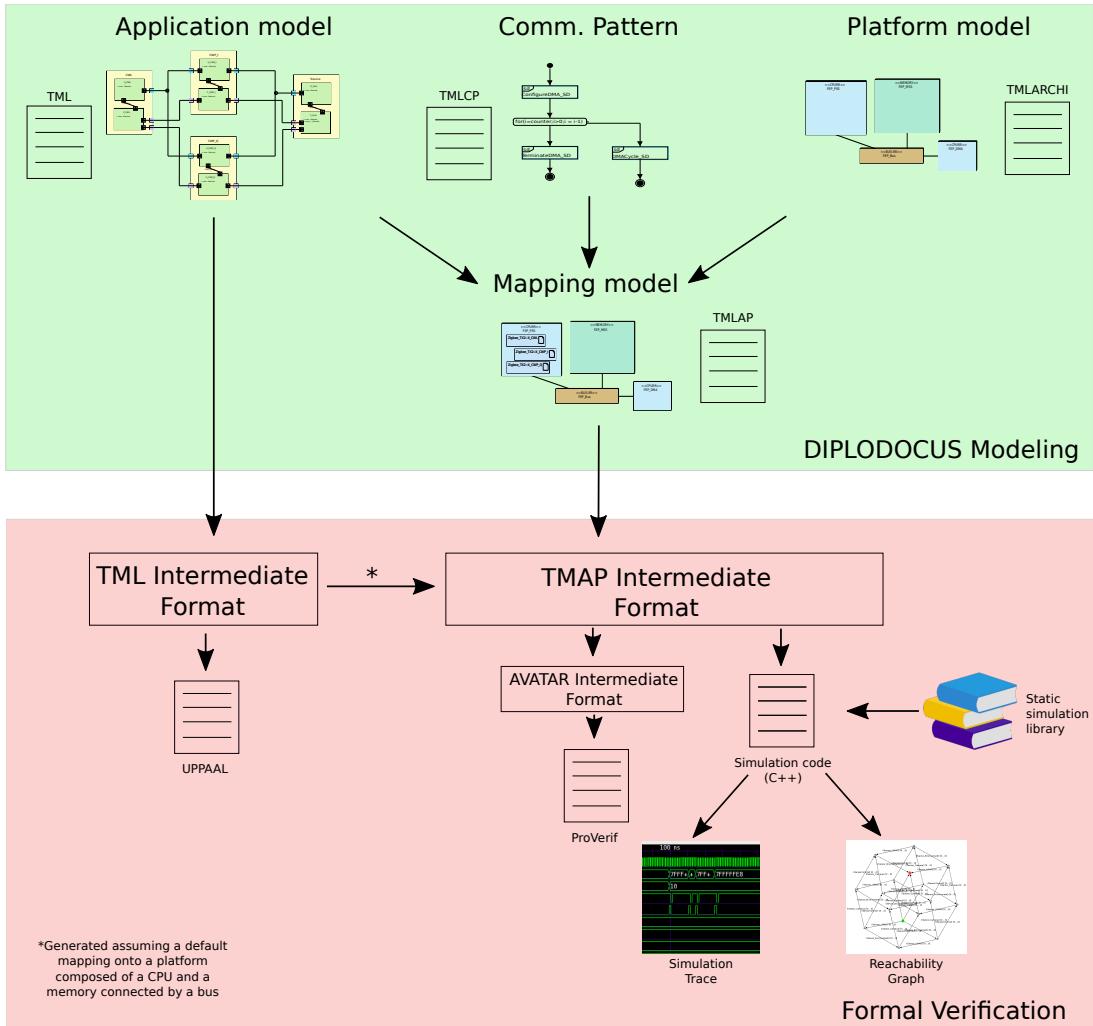
Formal verification is the act of proving whether a given property is satisfied by a system, or not, using formal techniques. In order to enable formal verification both after and before mapping, a formal semantics is provided to tasks (application model), communication between tasks (Communication Patterns) as well as to platform units [3][4]. This formal semantics is based on UPPAAL [23] and on our own formal verifier specification. Formal analysis can be performed in TTool/DIPLODOCUS at the push of a button and it does not require the user to have a background on formal techniques as the transformation from UML/SysML models to formal specification is completely transparent to the user.

**Formal Verification before mapping** UPPAAL [23] is a formal language based on communicating automata. A formal language is a language defined mathematically, relying on words defined with an alphabet. Constructs of the language have an operational semantics i.e. a precise mathematical meaning is given to each construct of the language. On the contrary, UML is not formally defined, since UML is defined using a document in plain text (English) and with an informal meta-model, and definitely not with a mathematical approach. That is, two persons reading the same UML diagram may have a different understanding of that diagram. Conversely, two persons reading the same UPPAAL description have exactly the same understanding of the system under design.

A design in DIPLODOCUS can be automatically translated to UPPAAL, using an algorithm implemented in TTool (Fig. 100). That algorithm being formally defined, and since a UPPAAL description is also formally defined, we can say that a DIPLODOCUS design is also a formal language. Therefore, two persons reading the same DIPLODOCUS model shall have exactly the same understanding of the system.

UPPAAL is not only the name of a language, but it is also the name of the toolkit that can handle specifications given in the UPPAAL language. In UPPAAL you can specify in a formal way properties you want to prove on your system (e.g., proving that the ZigBee transmitter never transmits an empty message): for each property, UPPAAL can answer "true", false, or "couldn't prove it". The "couldn't prove it" applies when UPPAAL is not able to prove the property. In this case, it is most likely that the system has so many execution traces that your computer can't compute all of them (e.g., lack of available memory). This situation is typically called *combinatory explosion*.

<sup>11</sup> We recall that Communication Patterns are transformed into equivalent TML tasks when the simulation code is generated for a given design. This transformation has been implemented in order to integrate CPs to the existing DSE infrastructure, without having to change the simulation engine

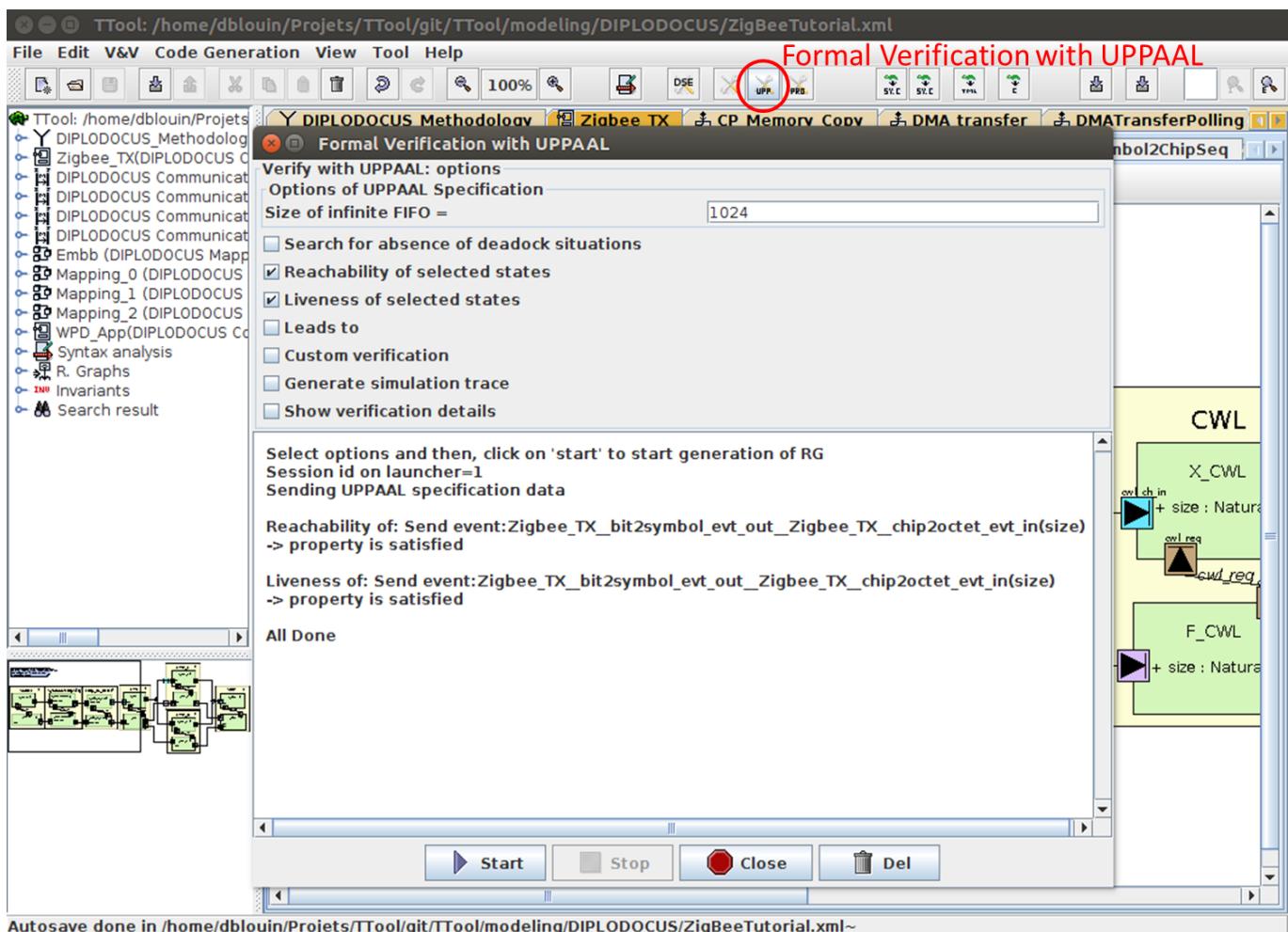


**Fig. 100.** The formal verification capabilities of TTool/DIPLODOCUS

**Pre-mapping formal verification with UPPAAL** The system properties that TTool/DIPLODOCUS can formally verify, jointly with UPPAAL, are *liveness* (i.e., whether a system state is reached in every possible execution) and *reachability* (i.e., whether there exists at least one execution reaching a given state). Currently, UPPAAL can be used for formal verification before mapping, to verify properties of the application model only. A formal translation of the semantics of Communication Patterns is part of our future work.

To proceed with pre-mapping formal verification, with UPPAAL, follow these steps:

1. Tag an operator of a task's activity diagram for which you want to study the reachability or the liveness. To do so, right click on the operator, and select **Check for accessibility/liveness**.
2. Check the syntax of the whole application model.
3. Click on the button highlighted in Fig. 101, **Formal Verification with UPPAAL**. A dialog box as shown at the center of Fig. 101 will open. The **Start** button launches the process that first translates the application model into an equivalent representation that is then taken as input by the UPPAAL engine to perform verification.



**Fig. 101.** The button and window to verify a DIPLODOCUS application model with UPPAAL

The dialog window center of Fig. 101 allows to specify which properties such as reachability and liveness of an application model's state shall be formally verified with UPPAAL. The results of this verification for the send-event operator `bit2symbol_evt_out(size)` of task `F_Symbol2ChipSeq` are shown in the dialog window. Both the reachability and the liveness properties of this operator are satisfied. This formally proves that, for each system execution, the block `Symbol2ChipSeq` always correctly transmits the control attribute `size` to block `Chip2Octet`.

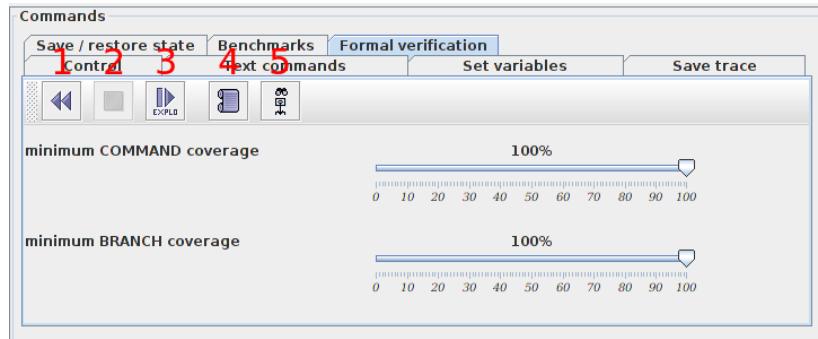
**Post-mapping formal Verification with the TTool verifier and simulator engine** Before mapping, tasks from the application models and activities from Communication Patterns have maximum concurrency between themselves.

However, once these models are mapped onto a target platform, concurrency is reduced as the execution of processing and transfer operations is constrained by buses and CPUs being shared among many tasks and activities. For example, two tasks mapped on the same CPU do not execute in parallel any more. An interesting property would be that formal traces obtained after mapping are a subset of formal traces obtained before mapping. In this case, it is crucial that traces obtained after mapping do not violate safety properties (e.g., absence of deadlock) that were present in models before mapping.

The simulator described in the previous sub-section also includes post-mapping formal verification capabilities. The simulation GUI is provided with a graphical bar that allows a user to select a minimum percentage of a mapping-model's state space that must be traversed during simulation (Fig. 102). The simulation engine thus includes model checking and static program analysis techniques that are used to compare and merge logically equivalent execution paths and/or recurring system states. The objective is to allow to take design decisions dynamically, at simulation run-time without regenerating the formal model used by the simulator. In fact, from a user's perspective, it is desirable to generate a formal model once and to subsequently traverse only the fraction of the model's space that concerns the system property under investigation. This fraction of the formal model's space may then be pruned with the aid of: (i) conventional coverage criteria (with respect to covered branches, statements, tasks, conditions, etc.), (ii) expertise provided by the user (e.g. potentially critical parts that are known a-priori) or (iii) heuristics that take into account (non-)functional properties.

Fig. 102 shows the **Formal verification** tab of panel **Commands**. The buttons listed below allow a user to view and navigate through the Reachability Graph (RG) generated by the simulation environment. A Reachability Graph (RG) captures all possible execution traces of a system. It represents all possible traces that can be obtained at simulation step, in a compact way, in form of a graph.

1. Reset simulation
2. Stop simulation
3. Run exploration
4. Analysis of last RG
5. View last RG

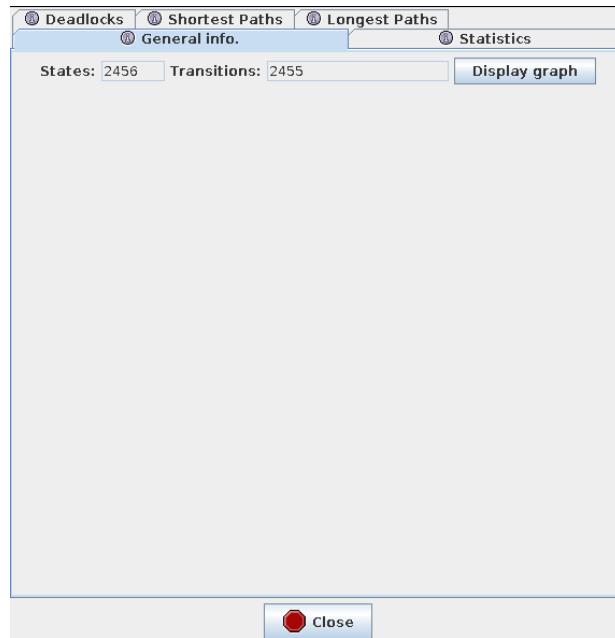


**Fig. 102.** The formal verification tab in the simulator Graphical User Interface

To analyze and view RGs, we recommend to install a graph viewer such as `dotty`, that is part of the the graph visualization software `Graphviz` [21].

**Example of post-mapping formal verification** As concluded by the analysis presented in [25], the throughput of the unslotted version of ZigBee varies with the number of data bits in the packet. For the case of the 2.4 GHz band that has been designed in this tutorial, a maximum throughput of 163 kbps can be achieved in a real system's implementation. In this tutorial, our models are designed to transmit messages whose size is 31 bytes (25 bytes payload and 6 bytes header). Such a throughput is equivalent to the transmission of 657 31-bytes messages/second. That is to say one message every  $1522\mu s$ . To verify if our design can satisfy this performance constraint, we can use the formal verification capabilities offered by TTool/DIPLODOCUS on the mapping model described in sub-section 7.8. To proceed with this formal verification, follow the steps below:

- We investigate the case of EMBB being implemented on a prototyping board. As explained at the end of subsection 7.4, we set the master clock frequency to 100 MHz, the Clock divider of each DSP unit to 1 and the Clock divider of the main CPU to 6.
- Check the syntax of the mapping model
- Generate the code for simulation
- Open the simulator’s GUI and connect to the simulator
- Switch to the Formal verification tab in the simulator’s GUI
- Click on button Run exploration
- Click on button Analysis of last RG
- You should now see the window shown in Figure 103. The graph contains 2456 states and 2455 transitions.



**Fig. 103.** Analyzing Reachability Graphs

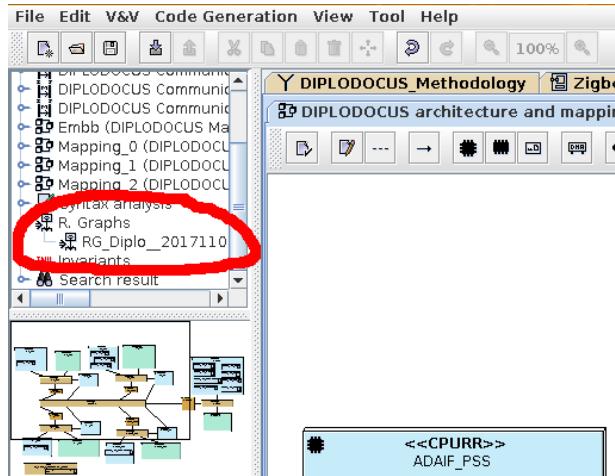
- Now, click on the ”Deadlocks” tab. Then, you should see the list of all transitions leading to states from which there is no output transitions. In our system, there is actually only one deadlock state (see Figure 104). From the transition leading to the deadlock state, we learn that the number of clock cycles to complete the processing of one packet is 6562 cycles. Since we known that the processor frequency is 100MHz divided by 6, we can deduce the time to compute one packet:  $time = 6562/10^8 * 6s = 397.372\mu s$ , which is less than  $1522\mu s$ .



**Fig. 104.** Analyzing Reachability Graphs: deadlock states

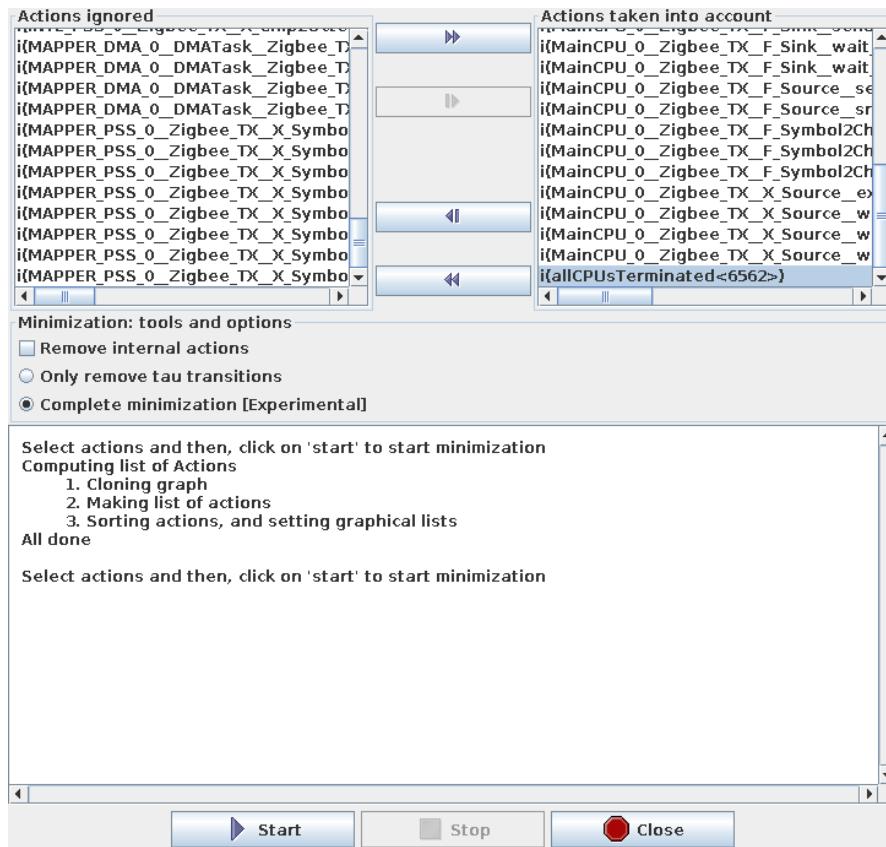
In the case where there are several transitions leading to a deadlock state, we can compute the minimum and maximum number of clock cycles to complete one execution of the system.

- Another way to formally analyze the system is to perform model-checking operations on the graph — which is not directly supported in TTool: you need to use an external toolkit to do this — or to reduce the graph to a reasonable size compatible with a visual analysis. Let’s now minimize the reachability graph to the actions that are performed on the main CPU only. First, go to the tree on the left, open the RG section, make a right click on the graph and select ”Minimize” (see Figure 105).



**Fig. 105.** Selecting a reachability graph previously generated

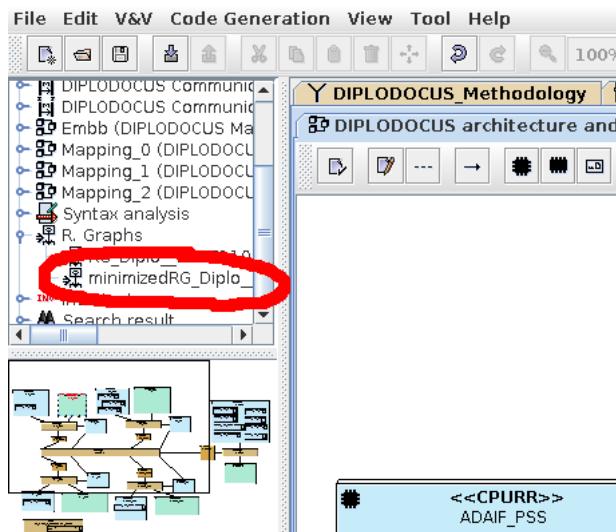
The minimization window should open. Move all transitions to the panel "Actions ignored" apart from the actions of the MainCPU and "allCPUsTerminated", see Figure 106. Then, click on start, and wait for the minimization to be completed. Then close the dialog window.



**Fig. 106.** Minimization window

The minimized graph is accessible in the left tree, see Figure 107.

Make a right click on the minimized graph, and select "Show". A RG similar to the one of Figure 108 should be displayed. The green state corresponds to the initial state, and the red state corresponds to the final state (or



**Fig. 107.** Accessing to the minimized RG

deadlock state). By "reading" this graph, we can easily verify that all the order of signal processing functions is always respected: Chip2Octet, CWL, and the two CWP functions.

**Post-mapping formal verification with ProVerif** Security analysis is performed with ProVerif, a verification tool operating on designs described in pi-calculus [28]. A ProVerif specification consists of a set of processes communicating on public and private channels. Processes can split to create concurrently executing processes, and replicate to model multiple executions (sessions) of a given protocol. Cryptographic primitives such as symmetric and asymmetric encryption or hash can be modeled through constructor and destructor functions. ProVerif assumes a Dolev-Yao attacker, which is a threat model in which anyone can read or write on any public channel, create new messages or apply known primitives.

ProVerif provides its user with the capabilities to query the confidentiality of a piece of data, the authenticity of an exchange, or the reachability of a state. Traces are generated for all possible execution paths. The tool then presents a result to the user that is either *true* if the property is verified, *false* if a trace that falsifies the property has been found, or *cannot be proved* if ProVerif failed in asserting or refuting the queried property.

To run ProVerif on a mapping, first run Syntax Analysis, and once the mapping is validated, click on the ProVerif Security Analysis Button. In the popup window shown in Figure 109, we input the location for the output file containing the ProVerif text specification, and the location of the installed ProVerif verifier. After clicking the 'Start' button, the Verification results are displayed. Figure 110 shows the verification results for the example described in Section 10.1, which shows reachable vs non-reachable states, and confidential and authentic data.

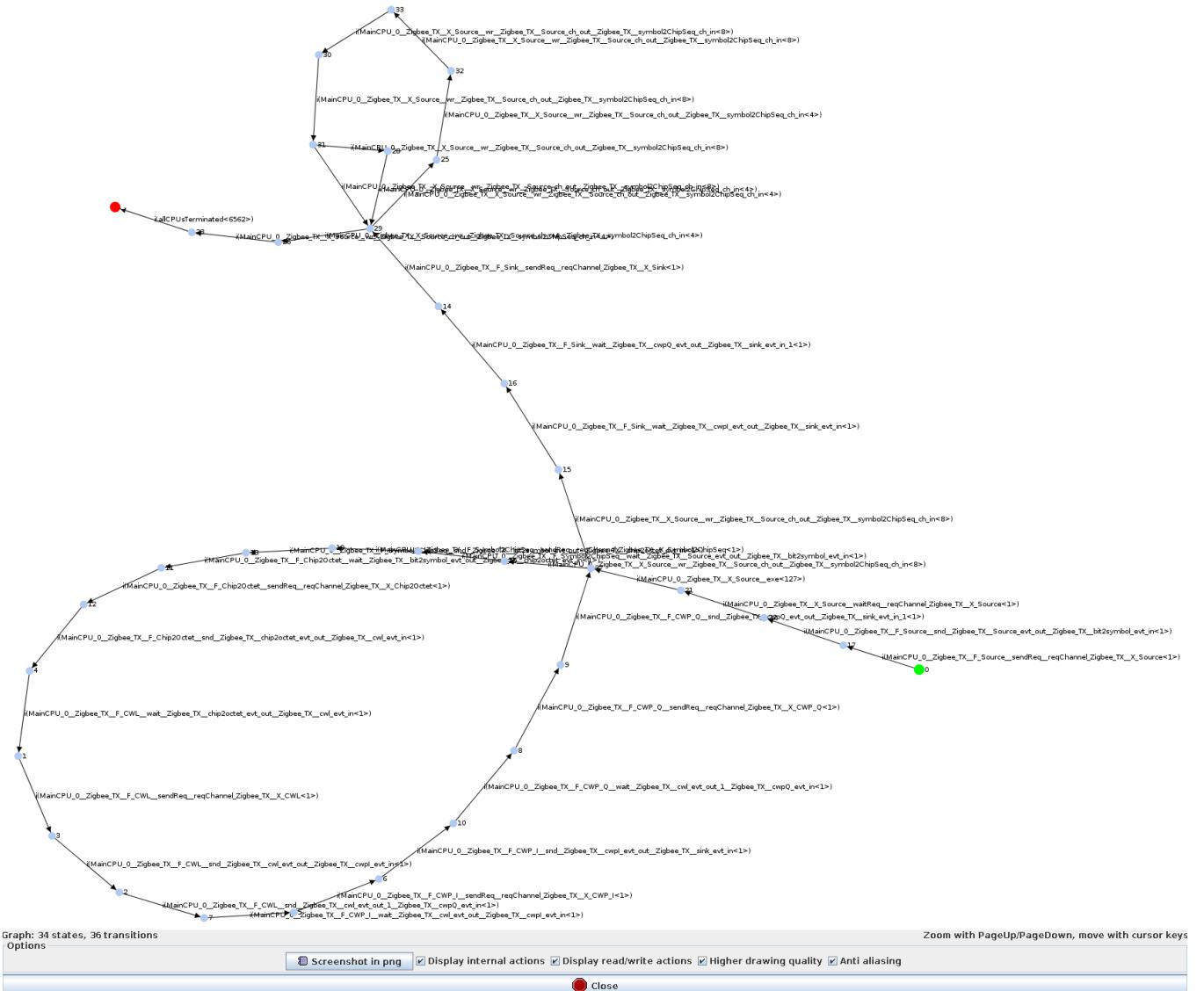
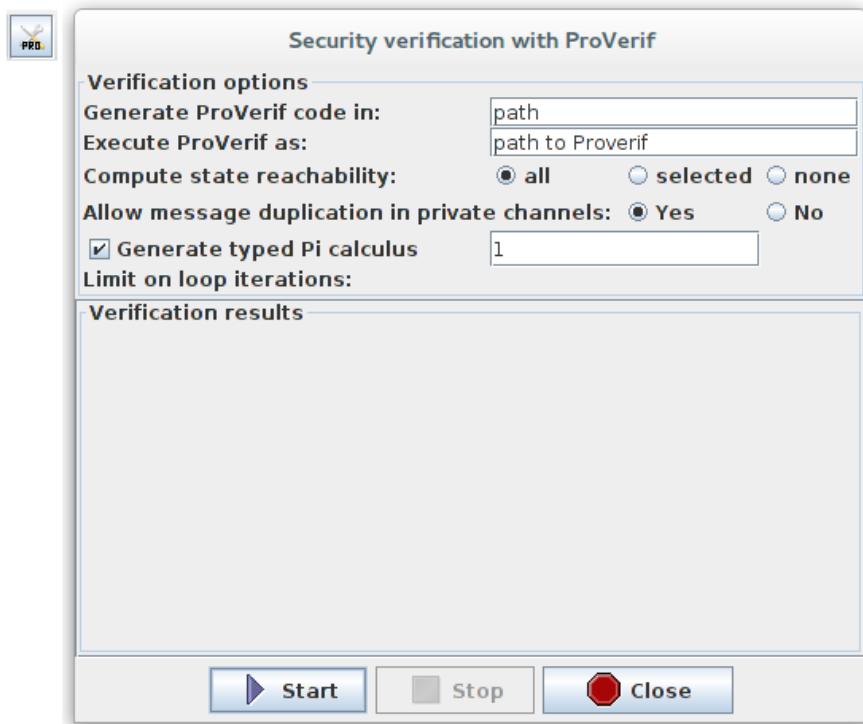
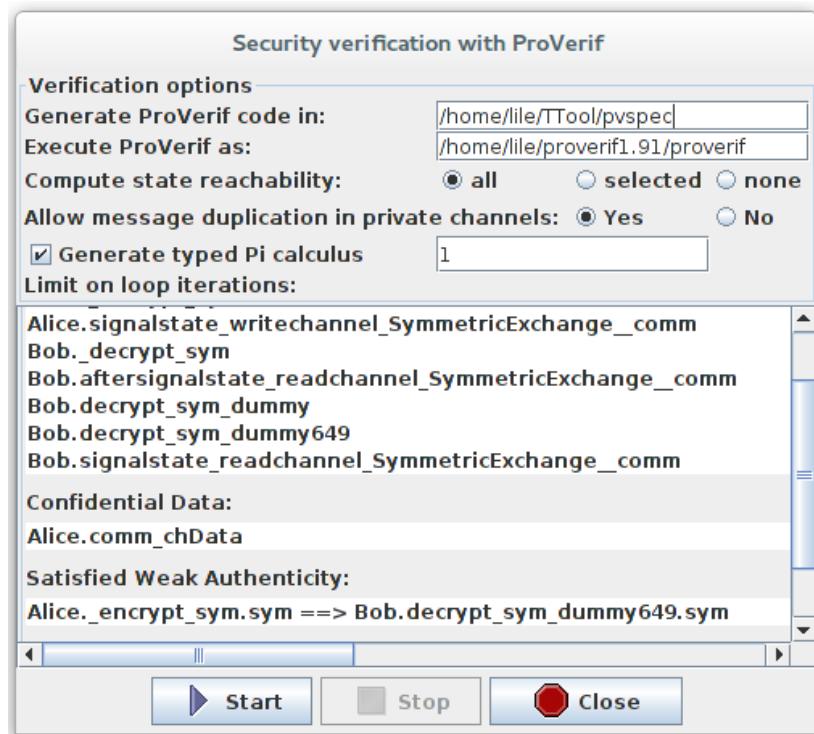


Fig. 108. Minimized RG



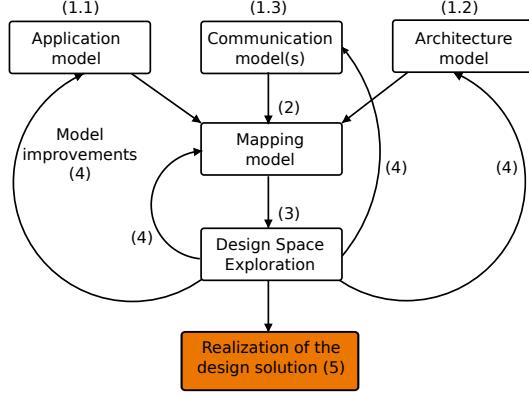
**Fig. 109.** The button and window to launch security verification with ProVerif.



**Fig. 110.** ProVerif verification results window.

## 9 Automatic Code Generation for Rapid Protoyping

Code generation for rapid prototyping (Fig. 111) aims at rapidly generating a system implementation in terms of either synthesizable hardware or compilable software. In the context of our work, we will discuss design implementation in terms of compilable software.



**Fig. 111.** The step of realizing a design solution, that is described in this section, in the context of the  $\Psi$ -chart design approach

Code generation from system-level models is challenging as target platforms are composed of a set of heterogeneous units (e.g., DSPs, CPUs, DMAs, Hardware Accelerators) with different characteristics such as Instruction Set Architecture, Application Programming Interface and memory organization. For a given functionality, it may be desirable to generate executable code for different target platforms. In this context, the key issue is how to efficiently add implementation details that are platform-specific, to system-level models that make use of high level abstractions in order to be platform-independent.

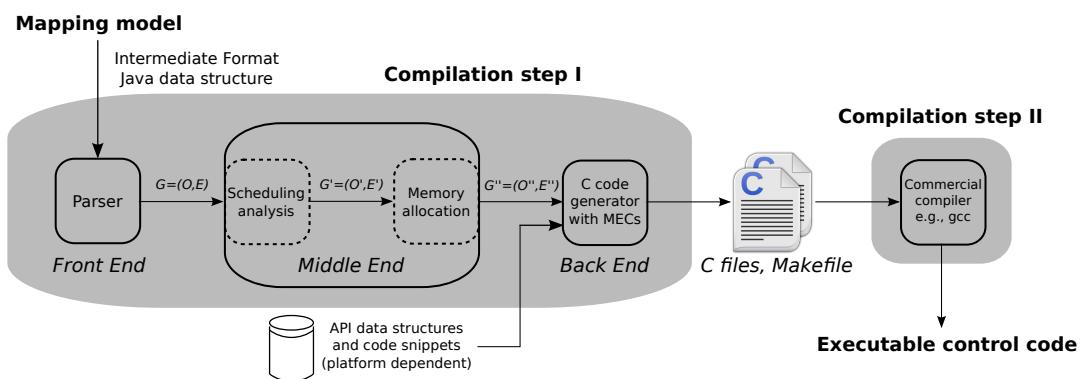
To address this issue, the approach that we propose (Fig. 112) is based on two separate compilation steps. First, an input mapping model is translated by a model-to-executable-code compiler into a C code representation that is compliant with a target platform's Application Programming Interface (API) and data structures. In this first compilation step, implementation details (e.g., data structures, register files) are included via a library of platform-specific entities called Model Extension Constructs (MECs). By linking to the compiler a library of MECs for a different platform (or for a different configuration for the same platform), the code generation process achieves the desired cross-platform portability. Secondly, this C code is given as input to a commercially available compiler (e.g., gcc, Turbo C) to produce an executable file.

In the rest of this section, we detail our implementation of the code generation process in Fig. 112. In this context, the model-to-executable-code compiler has been developed in Java in order to be easily plugged into the existing software architecture of TTool/DIPLODOCUS (Fig. 3). As this work is a first contribution that lays the ground for future developments, we specify here that our implementation is focused on signal-processing platforms. We also precise that, specifically to our implementation, the executable output file is a monolithic *application* that runs as a single process on top of the software stack (e.g., Board Support Package, Operating System) of a control processor in the target platform.

**The compilation process** The compilation process of Fig. 112 is an extension of the the code-generation engine first proposed in [16]. Compilation step I in Fig. 112 takes as input the equivalent representation of a mapping model from the Intermediate Format Java data structure of TTool/DIPLODOCUS (Fig. 3). It outputs a set of C files and a Makefile to automate the second compilation step<sup>12</sup>. In the output C files, processing and communication operations from the initial mapping model are transformed into three routines that contain initialization, execution and clean-up code. Additionally, a fourth routine, called *fire-rule*, is assigned to an operation to specify the logical dependencies that must be satisfied for its execution.

<sup>12</sup> This Makefile is expected to work in Linux

The front-end of our model compiler in Fig. 112 is a parser that checks the correctness and coherency of a mapping (e.g., the mapping of instances of a Communication Pattern must respect the topology specified in the platform model) and converts the IF Java data structure into a directed graph representation,  $G = (O, E)$ . In this graph, processing and communication operations constitute the vertexes  $o \in O$ . The edges  $e \in E$  in  $G$  represent dependencies between operations that are created based on the information entered by a user when mapping the models in the  $\Psi$ -chart. Subsequently, the compiler's middle-end takes as input the operation graph  $G$ , analyzes its schedulability and produces an annotated version  $G'$ , where edges and nodes are enriched with scheduling information.  $G'$  is then processed in order to allocate memory regions for input/output data of each processing and communication operation. This produces a second annotated graph,  $G''$ , that is transformed in C code by the compiler's back-end. The latter is a C code generator that also takes as input a library of data structures and code snippets that are compliant to the target platform's API. To cope with the heterogeneity of units in a target platform, the back-end relies on dedicated Model Extension Constructs (MECs). A MEC is associated to each annotated operation  $o \in O''$ , where  $O'' \in G''$ . It maps  $o$  to the code snippets and the data structures offered by the platform unit to which  $o$  had been bound in the initial mapping model.



**Fig. 112.** An overview of the two-step compilation process to generate the executable control code

**Scheduling of operations** Scheduling information is annotated in  $G' = (O', E')$  based on the *events* generated by the availability of data produced/consumed by operations  $o \in O$  ( $O \in G$ ), according to the Synchronous Data Flow Model of Computation. This Model of Computation (MoC) has been selected as our code generation engine currently targets radio signal-processing applications. As part of our future work, we will extend the scheduling of  $G$  to other MoCs. Our implementation of the scheduling analyzer favors an event-driven programming model rather than threads, as using threads and synchronization mechanisms would lead to rigid descriptions that are difficult to be scaled according to the different scenarios that can occur in data-dominated systems [13,14]. For instance, in a signal-processing system composed of multiple applications, in case one or more applications stop execution, it would be more difficult to re-synchronize their execution using threads [15].

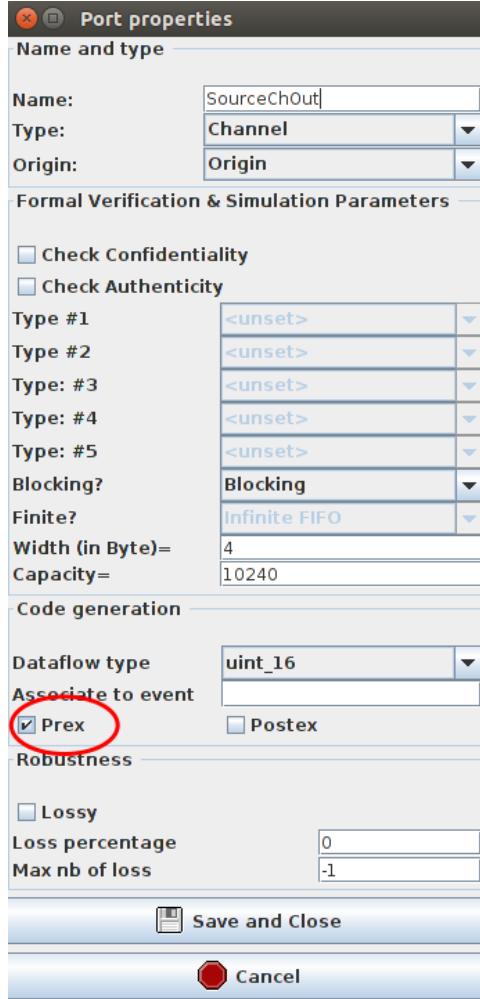
**Memory allocation** The compiler's middle-end in Fig. 112 allocates memory regions for operations according to the mapping information introduced by a user at step L1 (mapping of storage resources in Fig. 71). This results in a static allocation policy that we propose to extend to a more dynamic solution (i.e., the memory regions are selected by a memory manager at run-time), as part of our future work.

**Portability of the code-generation approach** Our framework addresses platforms where the scheduling of operations is centralized by a general-purpose control processor. The latter configures and dispatches the execution of operations to a set of physically distributed units (e.g., DSPs, DMAs), according to the events generated by the consumption/production of data. For a design project that includes multiple platforms with a centralized controller and distributed execution units, a library of MECs must be provided to compile mapping models to sets of code snippets and data structures that are compliant to different APIs. For each platform, dedicated MECs must be provided by re-using those from other projects as templates. To target architectures where both control and execution of operations are physically distributed onto different units, the C code generator must be adapted to produce multiple *applications*

that will each be executed by a different control processor. Therefore, synchronization primitives must also be added to coordinate the parallel execution of these applications.

### 9.1 Generating the code for the ZigBee transmitter

Before starting the process of code generation, we must mark the output data channel of the source block and the input data channel of the sink block, as **prex** (pre-execution) and **postex** (post-execution), respectively. To do so, double click on this channels and tick the **prex** and **postex** boxes as in Fig. 113 and Fig. 114. Channels marked as prex and postex are used by the code generation engine to build a graph representation of the application model,  $G'$ , from which the scheduling of operations is derived. The C code for rapid prototyping can be generated from a



**Fig. 113.** Mark the source output channel as **prex**

mapping diagram, after the Syntax analysis phase by clicking on the dedicated button shown in Fig. 115. Fig. 116 shows the graphical window that allows the start of C code generation. The directory where the C code is produced is automatically retrieved from the configuration file but can also be specified in the dialog window. The latter also allows a user to remove old files that were produced as a result of a previous generation process. The generated C code is an almost complete software implementation of the system that is derived from the information from the mapping model of TTool/DIPLODOCUS. The generated files are:

- ZigBee\_TX.h is the header file that declares the functions, variables, signals and buffers whose definition is provided in file ZigBee\_TX.c.

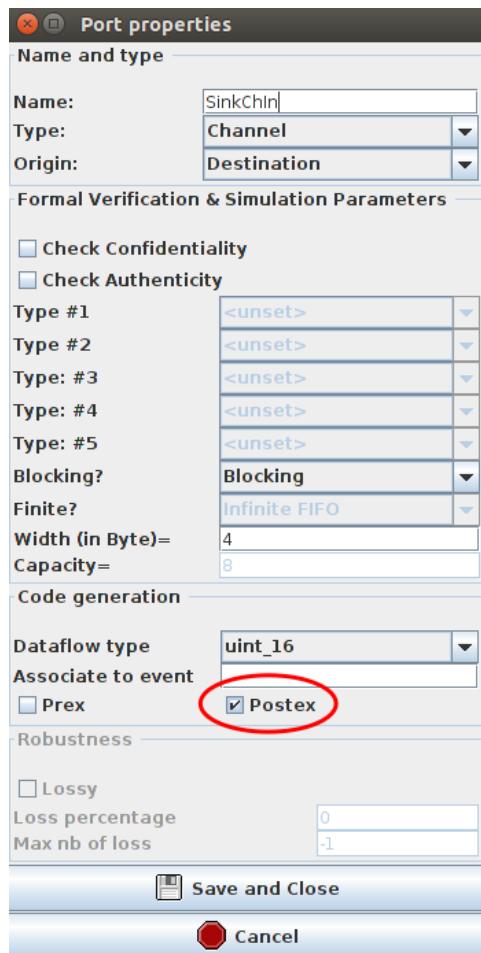
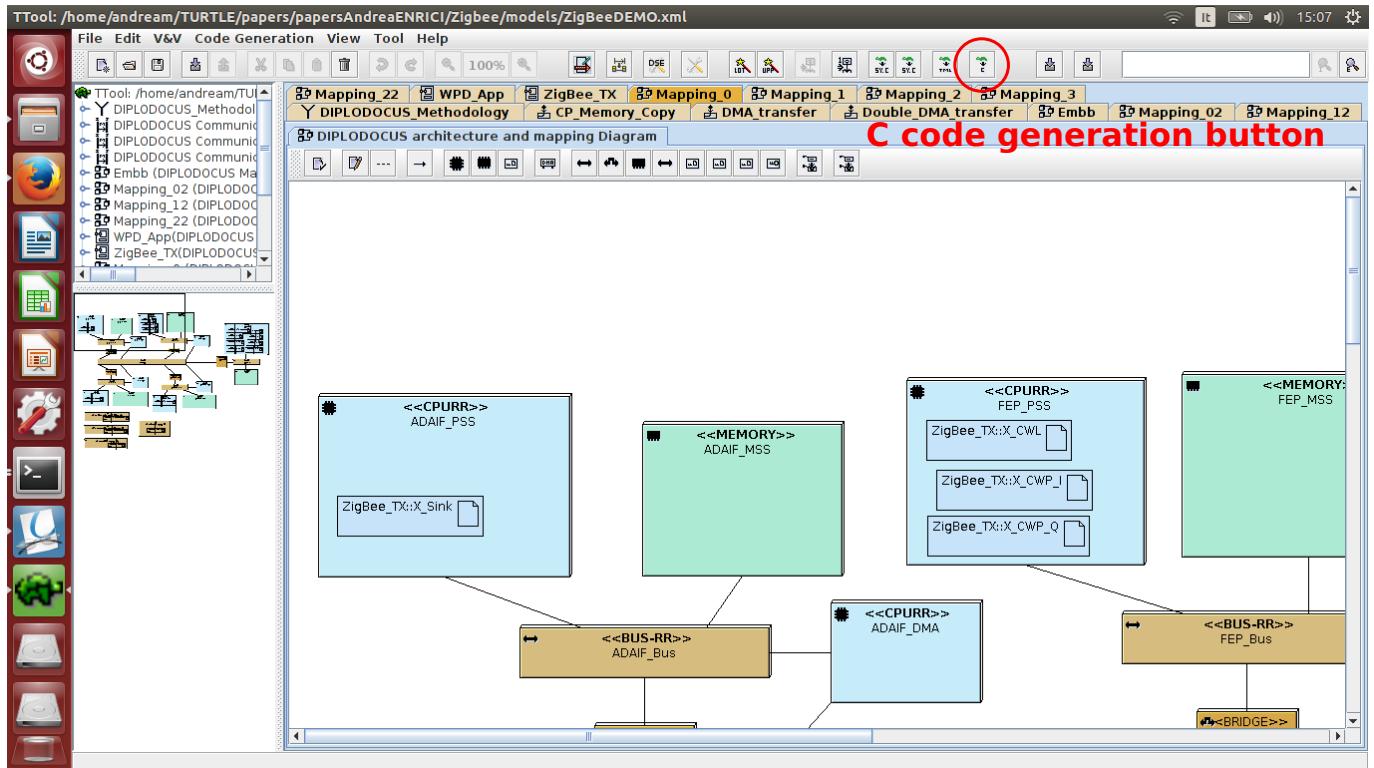
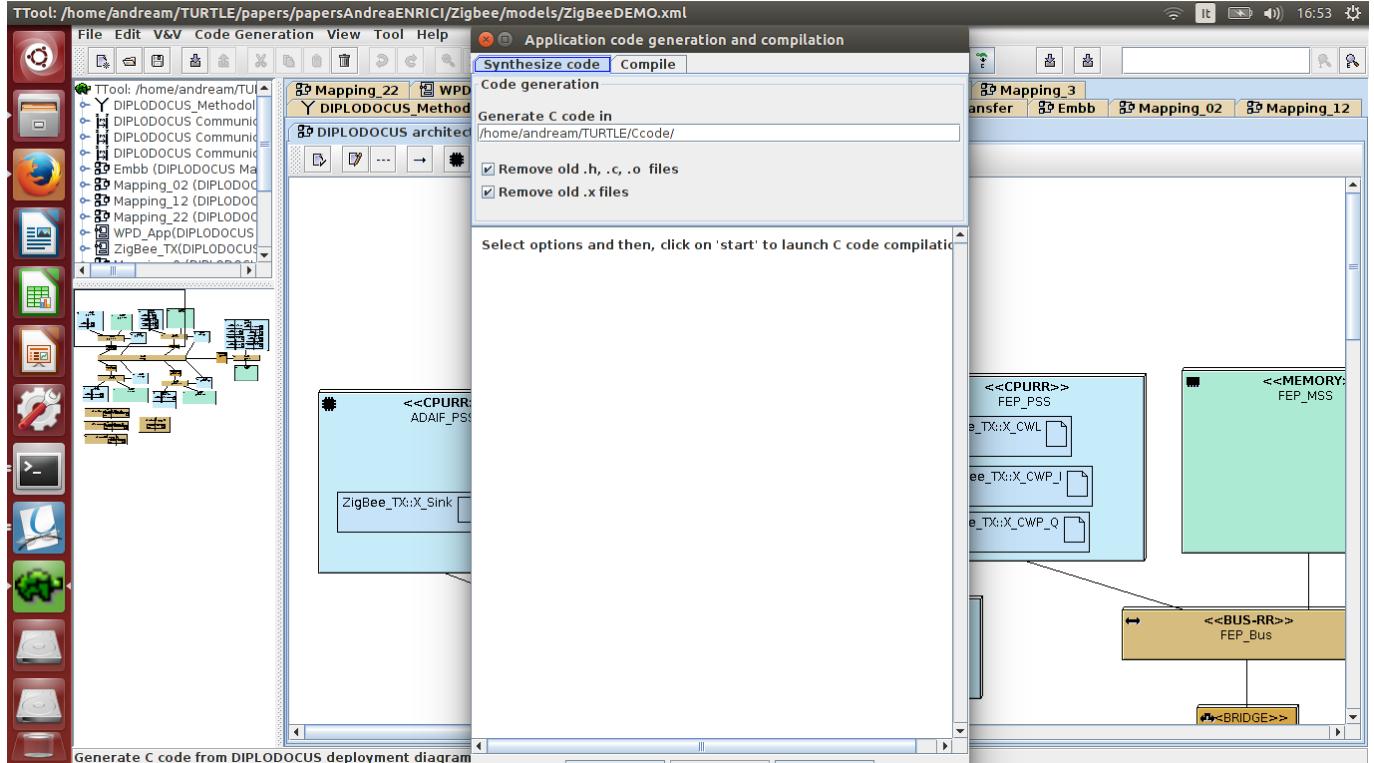


Fig. 114. Mark the sink input channel as postex



**Fig. 115.** The button to start the process of code generation for rapid prototyping



**Fig. 116.** Starting the C code generation for rapid prototyping

- ZigBee\_TX.c contains the definitions of functions, variables, signals and buffers used to configure and schedule the execution of the platform units. The main routine defined in this file is called `ZigBee_TX_exec`. It contains calls to the functions that initialize and cleanup the processing operations, the data transfers as well as signals and buffers. It also calls dedicated routines that set the initial values for the fire-rule routines. The kernel of routine `ZigBee_TX_exec` is the scheduler of the processing and transfer operations, whose pseudo-code is shown in Algorithm 1.
- ZigBee\_TX\_init.c initializes signals, buffers as well as the data structures that define the behaviour of both processing and transfer operations. Currently, memory allocation of data-blocks for the processing and transfer operations is left to the designer. We are working on automating this aspect as part of the future work.
- Makefile that automates the build process of an executable file. The Makefile is partially configured with information that is retrieved from the configuration file of TTool/DIPLODOCUS. Some configuration options and parameters are left to the user, e.g. the installation path of the code library that contains the target platform’s API.

---

**Algorithm 1** The pseudo-code of the Synchronous-Data-Flow (SDF) scheduler
 

---

```

1  execute_source_operation();
2  while sink_exit_rule() ≠ true do
3      for op ← 0 to NUM_OP – 1 do
4          ready ← operation_fire_rule[op];
5          if ready then eligible_ops.push(op);
6          eligible_ops.schedule_priority();
7          while eligible_ops.size() do
8              op ← eligible_ops.pop();
9              status ← operation_fire_rule[op];
10             if status then exit;
11         end while
12     end for
13     execute_sink_operation();
14 end while
  
```

---

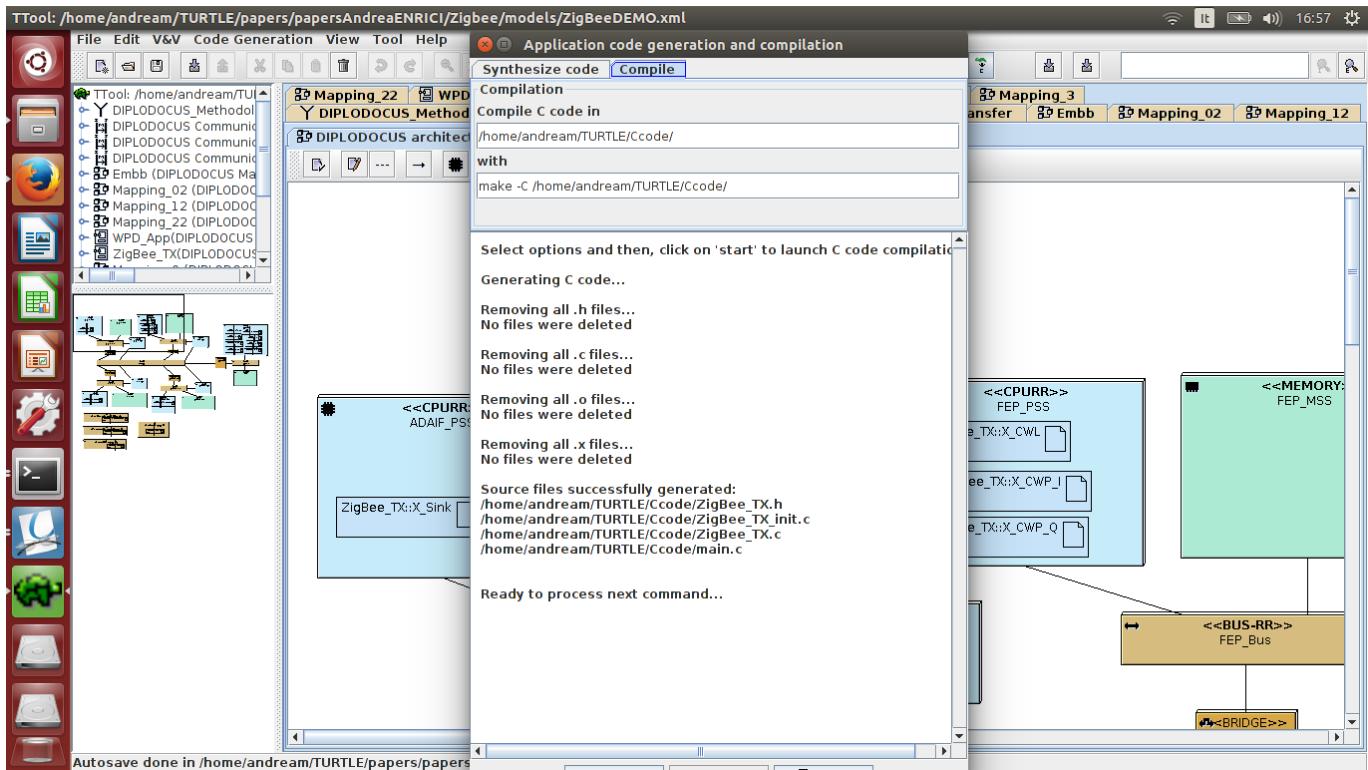
Click on the start button to produce the C code as shown in Fig. 117. Here, the dialog window allows the user to choose the location of the source files to be compiled and the command to be launched to produce an executable file. For the compilation process of Fig. 117 to complete successfully, the user must customize the Makefile that TTool produced and correctly enter the location of the libraries used by the MECs.

As mentioned above, the C code that is produced by Compilation step I in Fig. 112 is a skeleton of a complete implementation of the system described in UML/SysML models. More in detail, the code that is left to be written by the user is the code that defines the memory placement of the data-blocks produced/consumed by the processing operations. In the case of EMBB, these lines of code correspond to function calls that configure the transfer of data to/from the local Memory Sub-System from/to the Processing Sub-System of DSP units in EMBB (Fig. 42). Specifically for the ZigBee transmitter described in this tutorial, the platform-dependent code that is in charge of memory placement amounts to 202 lines. Out of these 202 lines of code, 19 lines can still be completely generated by the joint use of the  $\Psi$ -chart and Model Extension Constructs without the need for manual coding. On the other hand, 183 out of these 202 lines have to be *manually completed* by the user. By *manually completed*, we mean that the user has to manually insert input parameters in function calls. For instance, Listing 1.1 shows the Model Extension Construct for a Fast Fourier Transform of the Front End Processor. Here, the parameters that have to be manually inserted by the user can be seen at lines 20-27, 29-38, 43-47 (i.e., /\* USER TO DO: VALUE \*/ comments). Conversely, the joint use of the  $\Psi$ -chart approach and of MECs, allows the user to generate the code (70 lines) for data-transfers from Communication Patterns, without requiring the user to manually complete the code. The issue of the automatic synthesis of the code that manages the data memory allocation/deallocation from high-level models is part of our future work.

**Listing 1.1.** Example of Model Extension Construct for a Fast Fourier Transform for the Front End Processor (FEP) of EMBB

```

1 public FftMEC( String _ctxName, String inSignalName, String outSignalName )      {
2     name = "Fast Fourier Transform MEC";
3     exec_code = TAB + /*firm instruction*/ + CR +
4     "int status;" + CR + TAB +
5     "fep_set_l(&" + _ctxName + ", ((FEP_BUFFER_TYPE*)sig[" + inSignalName + "].pBuff)->num_samples);;" + CR + TAB +
6     "fep_set_qx(&" + _ctxName + ", ((FEP_BUFFER_TYPE*)sig[" + inSignalName + "].pBuff)->bank);;" + CR + TAB +
7     "fep_set_bx(&" + _ctxName + ", ((FEP_BUFFER_TYPE*)sig[" + inSignalName + "].pBuff)->base_address);;" + CR + TAB +
8     "fep_set_tx(&" + _ctxName + ", ((FEP_BUFFER_TYPE*)sig[" + inSignalName + "].pBuff)->data_type);;" + CR + TAB +
9     /*start execution*/ + CR + TAB +
10    "status = fep_do(&" + _ctxName + ");;" + CR;
11
12
13    init_code = "***** INIT " + _ctxName.split("_ctx")[0] + " *****/" + CR +
14    "void init_" + _ctxName.split("_ctx")[0] + "(void){;" + CR + TAB +
15    "fep_ctx_init(&" + _ctxName.split("_ctx")[0] + ", (uintptr_t) fep_mss );;" + CR + TAB +
16    // initialize context" + CR + TAB +
17    "fep_set_op(&" + _ctxName + ", FEP_OP_FFT );;" + CR + TAB +
18    "fep_set_r(&" + _ctxName + ", (uint64_t)/* USER TODO: VALUE */);;" + CR + TAB +
19    // X vector configuration => Zk=Y[Xi]" + CR + TAB +
20    "fep_set_wx(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
21    "fep_set_sx(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
22    "fep_set_nx(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
23    "fep_set_mx(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
24    "fep_set_px(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
25    "fep_set_dx(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
26    "fep_set_vrx(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
27    "fep_set_vix(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
28    // Y vector configuration" + CR + TAB +
29    "fep_set_by(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
30    "fep_set_qy(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
31    "fep_set_my(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
32    "fep_set_ny(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
33    "fep_set_sy(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
34    "fep_set_py(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
35    "fep_set_wy(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
36    "fep_set_ty(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
37    "fep_set_vry(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
38    "fep_set_dy(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
39    // Z vector addressing configuration" + CR + TAB +
40    "fep_set_qz(&" + _ctxName + ", (uint64_t) ((FEP_BUFFER_TYPE*)sig[" + outSignalName + "].pBuff)->bank);;" + CR +
41    TAB +
42    "fep_set_bz(&" + _ctxName + ", (uint64_t) ((FEP_BUFFER_TYPE*)sig[" + outSignalName + "].pBuff)->base_address);;" +
43    CR + TAB +
44    "fep_set_tz(&" + _ctxName + ", (uint64_t) ((FEP_BUFFER_TYPE*)sig[" + outSignalName + "].pBuff)->data_type);;" +
45    CR + TAB +
46    "fep_set_wz(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
47    "fep_set_ri(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
48    "fep_set_sz(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
49    "fep_set_nz(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
50    "fep_set_mz(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
51    // Operation configuration" + CR + TAB +
52    "fep_set_sma(&" + _ctxName + ", (uint64_t) /* USER TODO: VALUE */);;" + CR + TAB +
53    "}" + CR;
54    cleanup_code = "fep_ctx_cleanup(&" + _ctxName + "_ctx);";
55 }
```

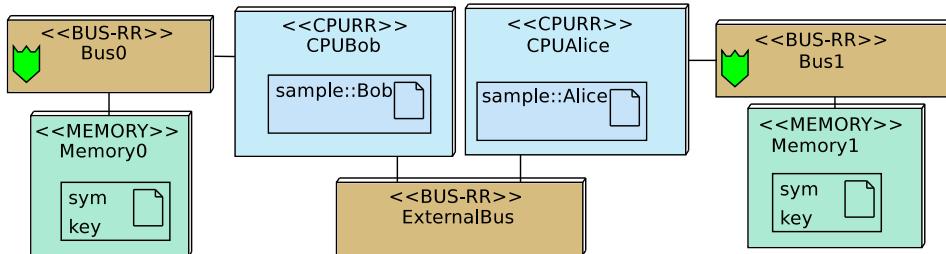


**Fig. 117.** Compiling the C code that has been automatically generated in Fig. 116

## 10 Analysis of security properties

In this section we present a case study that describes the analysis of security properties of embedded systems with TTool/DIPLODOCUS.

Even if we do not operate on named data or concern ourselves with values in DIPLODOCUS, it is still important that we consider security in the selection of a mapping. Encryption and decryption operations occupy computation cycles, or may add additional bits to data sent along a channel. To accurately analyze the final secured system's performance and safety properties, we use the security operators and tags to abstractly model all security-related operations.

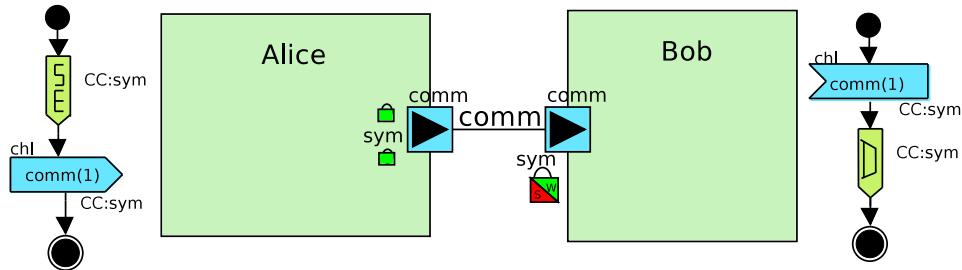


**Fig. 118.** Simple secured message exchange architecture

### 10.1 Symmetric Encryption

Here, we present how to model the basic exchange of Alice and Bob communicating across a public bus as shown in Fig. 118, model SysMLSec/AliceAndBobHW.xml. We assume they have pre-shared a secret key. We name this

Cryptographic Configuration 'sym', for Symmetric Encryption. Alice encrypts a message with the secret key under configuration 'sym', and sends it along a channel. To denote that Alice sends secured data, the channel 'comm' is tagged with 'sym' as shown in Fig. 119. Bob receives the encrypted data from the channel and then decrypts it.

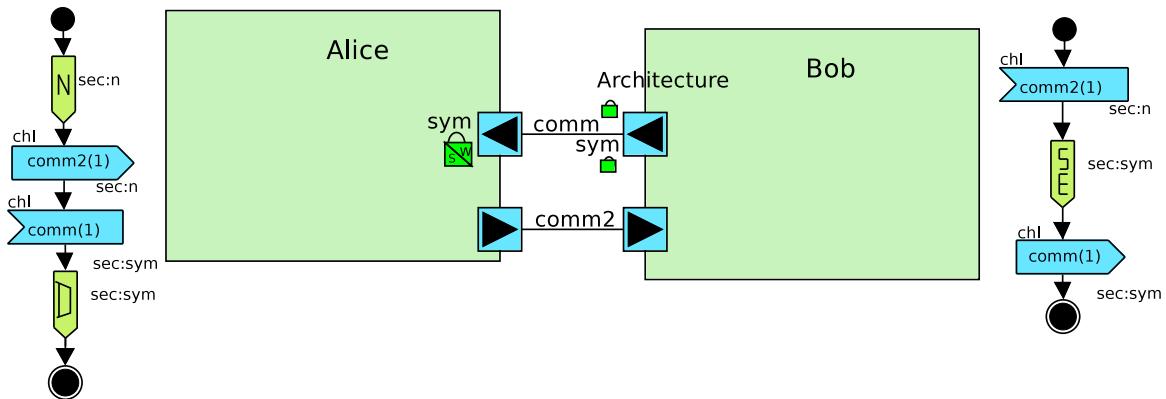


**Fig. 119.** Simple secured message exchange

We note that this exchange verifies confidentiality and weak authenticity of the secret data of 'sym', but not strong authenticity. An attacker could capture the exchange and replay it, and Bob would not be able to determine that the message came from an attacker and not Alice. If we wish to preserve strong authenticity, nonces must be added to the message before encryption.

## 10.2 Nonces

Nonces can be concatenated to messages to ensure strong authenticity and prevent replay attacks. Nonces are first forged as a Cryptographic Configuration. To state that a message will use a nonce, in the Cryptographic Configuration window, select the nonce name. As shown in Fig. 120, Bob sends Alice his nonce, then Alice concatenates it to her message and encrypts the new message. Bob then decrypts the message and verifies it contains his sent nonce. (Note that we allow a nonce to be used for more than one message)

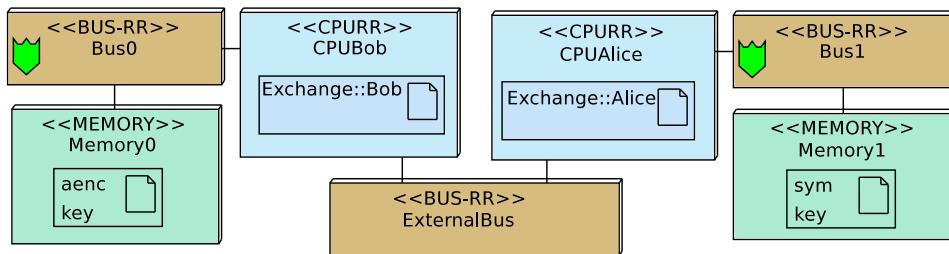


**Fig. 120.** Message exchange with nonce

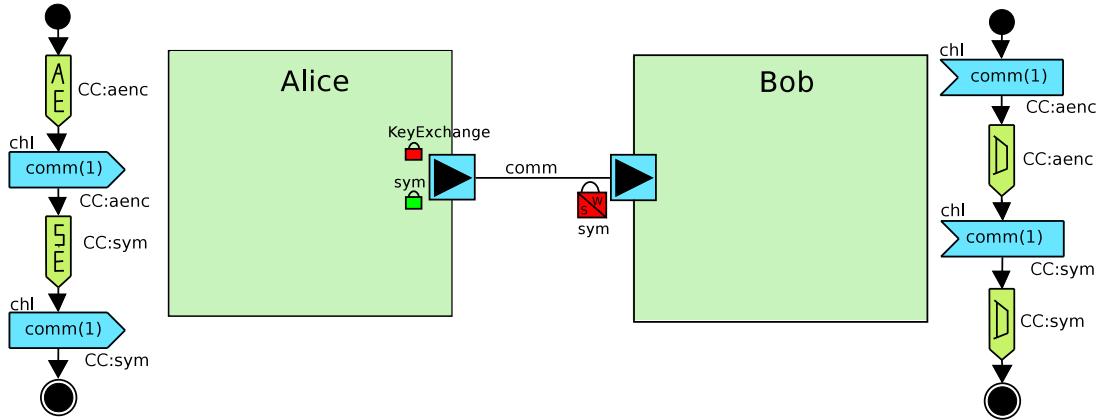
## 10.3 Key exchange

Key exchange can be modeled, or keys can be automatically distributed and their distribution be implicit. ProVerif returns an error if a task attempts to use a key which is either not sent or not mapped to an accessible memory.

We present a simple example of modeling key exchange using security operators. The architecture with mapped keys is shown in Fig. 121 (for asymmetric encryption, the mapped key is assumed to be the private key). Alice and Bob wish to communicate across a public channel. To send a message to Bob, first Alice encrypts a secret key with Bob's public key, and sends it to Bob along a public channel as shown in Fig. 122.



**Fig. 121.** Key exchange architecture with mapped keys



**Fig. 122.** Key exchange protocol

Next, Bob decrypts Alice's message with his private key to recover the secret key. To show that Bob is able to use the secret key, we model a subsequent message exchange.

Alice sends a message encrypted with the secret key to Bob. Using the secret key, Bob decrypts Alice's newest message and recovers the original message.

If Bob uses the secret key to send a message to Alice, however, this key exchange protocol does not preserve a secret message. An attacker could pretend to be Alice and send an encrypted secret key to Bob, and then Bob will encrypt his secret message with the attacker's secret key and send it.

TTool assumes that the only accessible keys are along private buses, as accessing a key along a public bus would be a security violation. While the ProVerif specification will still be generated assuming each task has access to the key, a warning debug message will be printed.

## 10.4 MAC

If it is necessary to ensure the authenticity instead of confidentiality of messages, we may instead calculate a Message Authentication Code to concatenate onto the message, which the receiver can then use to verify that the MAC matches the received message.

For example, as shown in Figure 123 (using the Architecture of Figure 118), Alice calculates a MAC for a message and then concatenates the MAC onto the message before sending it. After Bob receives the message, he splits the message into the original message and the MAC, and then verifies the MAC to ensure that the message has been received unaltered.

## 10.5 Automated Security Generation

While Cryptographic Configurations can be manually handled by the designer, it is also possible to automatically generate these security elements. Based on security requirements provided by the designer, our toolkit automatically adds appropriate security for each channel whose data must be secure. Added security may be in the form of MACs, nonces, Symmetric Encryption, etc. This automated encryption adds a basic estimation of security, which the designer can later modify.

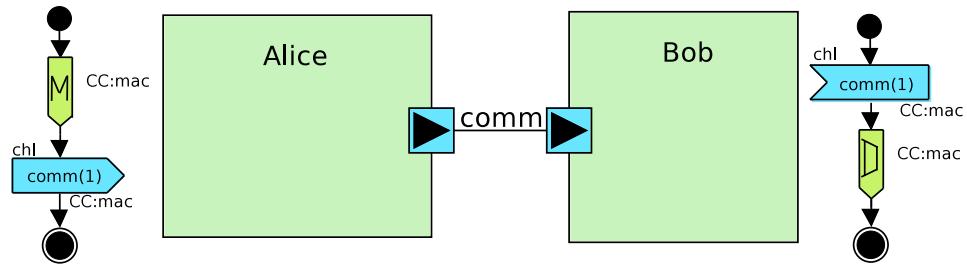


Fig. 123. MAC verification protocol

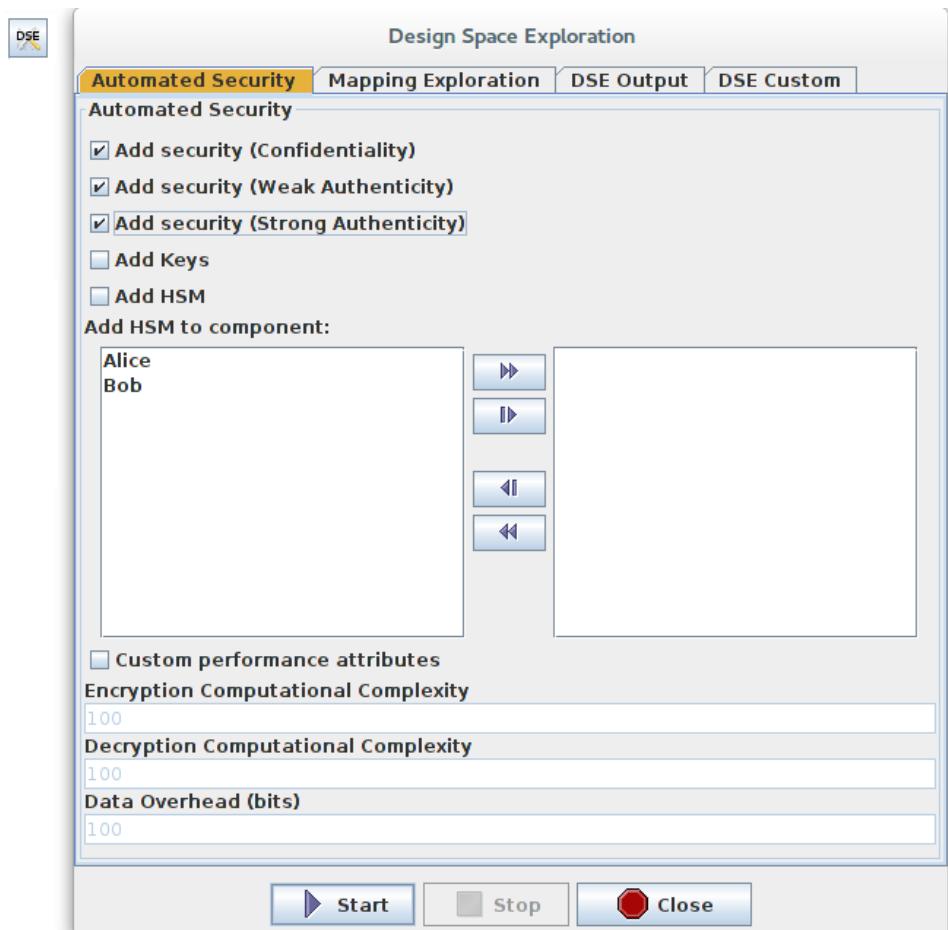
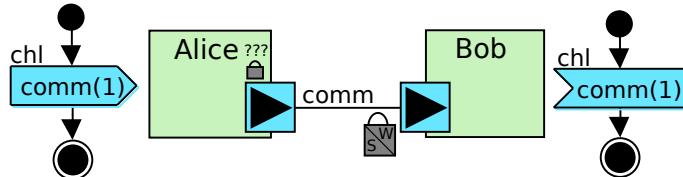


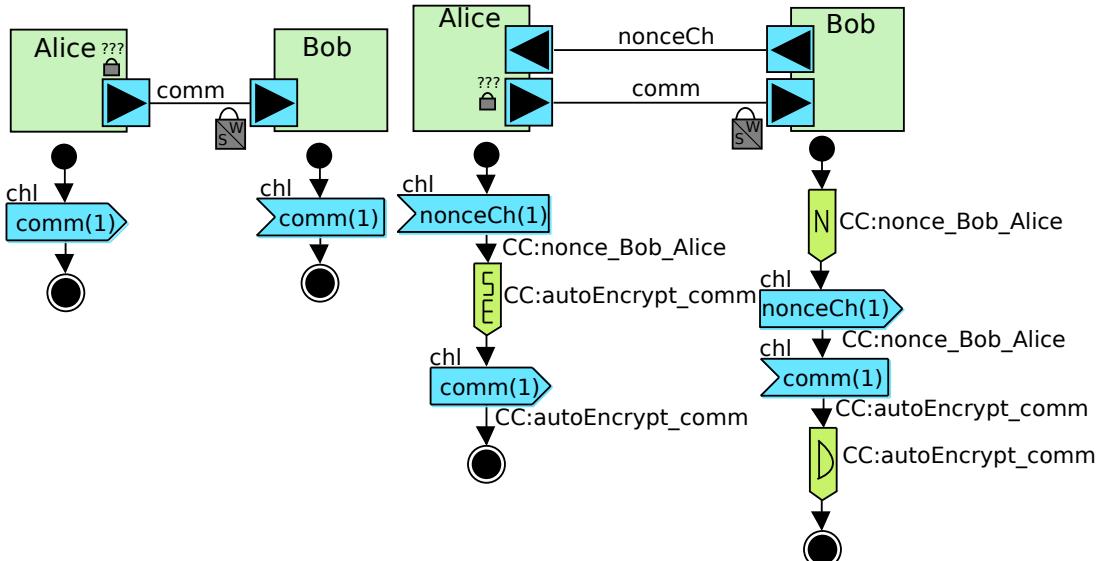
Fig. 124. Button and window for Automatic Security Generation

In our example in Figure 125, data sent along Channel 'comm' has been marked to be required confidential and authentic. However, the tasks are mapped to communicate across an insecure bus. After syntax analysis of a mapping, we open the Automatic Security window and select that we wish to preserve all security properties as shown in Figure 124, and then click 'Start'.

Figure 126 shows the models for Bob and Alice before and after security is generated. To ensure strong authenticity (prevent replay attacks), Alice and Bob must exchange a nonce before each message exchange.



**Fig. 125.** Application model for Security Generation Example



**Fig. 126.** Unsecured vs Secured Application Models with Automatic Generation

## 11 Conclusion

We have presented a tutorial to help users learn how to use TTool/DIPLODOCUS for the development of modern embedded systems through electronic design automation making use of model-based engineering. The full design of a ZigBee transmitter from modeling to automatic code generation including formal verification, simulation and design space exploration have been presented.

For more information, please visit <https://ttool.telecom-paristech.fr/> or contact any of the authors of this tutorial.

## Appendix 1.A Formal description of Communication Patterns

The following tuple provides a formal description of the UML/SysML diagrams and operators that compose a Communication Pattern.

$$\mathcal{CP} = (\mathcal{M}_{CP}, \mathcal{AD}_{CP}, \mathcal{SD}_{CP})$$

- $\mathcal{M}_{CP}$  is the main Activity Diagram (interface) of a Communication Pattern  $\mathcal{CP}$
- $\mathcal{AD}_{CP}$  is the set of Activity Diagrams that are referenced in the entire Communication Pattern  $\mathcal{CP}$
- $\mathcal{SD}_{CP}$  is the set of Sequence Diagrams that are referenced in the entire Communication Pattern  $\mathcal{CP}$

An Activity Diagram  $\mathcal{AD}$  is defined as the following tuple:

$$\mathcal{AD} = (\mathcal{R}_{SD}, \mathcal{R}_{AD}, \mathcal{C}_{op}, N, L)$$

- $\mathcal{R}_{SD}$  is the set of Sequence Diagrams referenced by  $\mathcal{AD}$ . A reference to a Sequence Diagram  $r \in \mathcal{R}_{SD}$  is considered as a node  $n \in N$  that has one incoming and one outgoing edges.
- $\mathcal{R}_{AD}$  is the set of Activity Diagrams referenced by  $\mathcal{AD}$ . A reference to an Activity Diagram  $r \in \mathcal{R}_{SD}$  is considered as a node  $n \in N$  that has one incoming and one outgoing edges.
- $\mathcal{C}_{op}$  is the set of control operators that are used to compose the references to other diagrams. A control operator  $c \in \mathcal{C}_{AD}$  can be of type  $c \in \{\text{parallelism, sequence, choice, iteration, start, final}\}$ .  $\text{start}$  is the start node (symbol) of the Activity Diagram.  $\text{final}$  is the end node (symbol) of a *path* (defined in Property 5) within an Activity Diagram.
- $N$  is the set of nodes that compose the Activity Diagram. A node  $n \in N$  is either a reference to a diagram  $r \in \{\mathcal{R}_{SD}, \mathcal{R}_{AD}\}$  or a control operator  $c \in \mathcal{C}_{op}$ .
- $L$  is a set of links (edges). Each link  $l \in L$  interconnects a pair of nodes  $n_1, n_2 \in N$  with the following notation:

$$l_{n_1, n_2} = n_1 \rightarrow n_2$$

A Sequence Diagram  $\mathcal{SD}$  is defined as the following tuple:

$$\mathcal{SD} = (\mathcal{I}_{SD}, E, \mathcal{M}_{SD}, \mathcal{A}_{SD}, \prec, \mathcal{V}_{I_{SD}})$$

- $\mathcal{I}_{SD}$  is the set of instances that are used to describe the components of a communication protocol. An instance  $i \in \mathcal{I}_{SD}$  can be of type  $i \in \{\text{controller, transfer, storage}\}$ .
- $E_i$  is a set of events that compose the lifeline of an instance  $i \in \mathcal{I}_{SD}$ . Each event  $e \in E$  can be one of type  $e \in \{SND_m, RCV_m, ACT_a\}$ , where:
  - $SND_m$  is the dispatch (send) of a message  $m \in \mathcal{M}_{SD}$
  - $RCV_m$  is the reception of a message  $m \in \mathcal{M}_{SD}$
  - $ACT_a$  is the occurrence of an action  $a \in \mathcal{A}_{SD}$
- $\mathcal{M}_{SD}$  is the set of parameterized messages that are exchanged by instances  $\mathcal{I}_{SD}$ . A message  $m \in \mathcal{M}_{SD}$  is part of a library that is composed of messages  $m \in \{\text{Read(), Write(), TransferRequest(), TransferTerminated()}\}$ . To ease the transformation of CPs in source code for DSE and for the control code synthesis, we currently consider only synchronous messages in  $\mathcal{M}_{SD}$ . We envisage to extend the semantics of messages to the asynchronous case in our future work.
- $\mathcal{A}_{SD}$  are the actions performed by instances  $\mathcal{I}_{SD}$  of type *controller* on variables  $\mathcal{V}_{I_{SD}}$  of a Sequence Diagram  $s \in \mathcal{SD}$ . As part of these actions, a parameterized timing function called *wait()* is also available.
- $\prec$  is a total order relation of events  $e \in E_i$ .
- $\mathcal{V}_{I_{SD}}$  is a set of user attributes (variables) of an instance  $i \in \mathcal{I}_{SD}$ . A variable  $v \in \mathcal{V}_{I_{SD}}$  can be of type  $v \in \{\text{integer, boolean, address}\}$ . It can be assigned a value in Sequence Diagrams  $s \in \mathcal{SD}$ . This value is of type *read-only* in the guards of the *choice* control operator, within an Activity Diagram.

In order to transform a Communication Pattern into an equivalent representation in terms of code that can be executed for DSE and for prototyping (see Section 9), we specified the following semantic properties that define a *well-formed* CP:

1. **Property 1. Non-modeling of returned data:** data that is returned upon the reception of a message is not modeled as we assume it to be implicit. For instance, data that is returned upon the issue of a *Read()* message is not modeled.

2. **Property 2. Access to attributes:** variables in  $\mathcal{V}_{I_{SD}}$  are read-only in Activity Diagrams. They are initialized in the Sequence Diagrams  $\mathcal{SD}$  and their value can be changed by actions  $\mathcal{A}_{SD}$  only. Only variables of type *int* and *boolean* can be used to govern the execution of control operators  $\mathcal{C}_{AD}$ . Variables of type *address* are used for automatic generation of the executable system control code.
3. **Property 3. Active instances:** instances of type *controller* are the only type of active instances. Both *controller* and *transfer* instances are allowed to both send and receive messages. However, *transfer* instances are only allowed to forward incoming messages. Instances of type *storage* are only allowed to receive messages.
4. **Property 4. Starting diagram:** in any Activity Diagram of a Communication Pattern, the starting symbol must always be followed by a reference to a non-empty Sequence Diagram.
5. **Property 5. Path:** we define a (generic) path as a set of interconnected nodes that terminate with the *final* node, as follows:

$$\begin{aligned} \text{path} = & (n_1, l_{n_1, n_2}), \{(n_i, l_{n_i, n_{i+1}})_{i:2 \rightarrow m-1}^+\}, (n_m, l_{m, final}, final), \\ & \text{with } n \in N, l \in L \end{aligned}$$

6. **Property 6. Complete path:** in any Activity Diagram of a Communication Pattern, a continuous path must interconnect the *start* and the *final* nodes via a set of control operators and references to diagrams in  $N$  and links  $l \in L$ . It is defined as follows:

$$\begin{aligned} \text{complete\_path} = & \\ & (start, l_{start, n_1}), \{(n_i, l_{n_i, n_{i+1}})_{i:1 \rightarrow m-1}^+\}, \\ & (n_m, l_{m, final}, final), \text{ with } n \in N, l \in L \end{aligned}$$

where the repetition operator  $(...)^+$  defines one or multiple occurrences of the content enclosed by the parenthesis. The above path can be developed as:

$$\text{complete\_path} = \{start, l_{start, n_1}, n_1, l_{n_1, n_2}, n_2, l_{n_2, n_3}, \dots, l_{n_m, final}, final\}$$

where  $m$  defines the length of the path in terms of interconnected nodes.

7. **Property 7. Parallelism control operator:** the parallelism operator is considered as a single node  $n \in N$ . It is composed of a fork and a join bars that are used as delimiters to, respectively, fork and join the execution of  $k$  branches of sequentially interconnected diagrams.

$$\begin{aligned} \text{parallelism} = & \\ & \text{fork } \{(l_{fork, n_1}, n_1)^t (l_{n_i, n_{i+1}}, n_{i+1})_{i:1 \rightarrow m}^t (n_m, l_{n_m, join})^t\}^{t:1 \rightarrow k} \text{ join} \end{aligned}$$

Above, index  $t$  is used to label each parallel interconnected branch.

8. **Property 8. Choice control operator:** the choice control operator is composed of one incoming and  $k$  outgoing edges, one for each outgoing branch. Each outgoing branch is labeled by a  $guard_k$  that specifies a boolean condition for the corresponding branch to be executed.

$$\text{choice} = \{guard_i, path_i\}_{i:1 \rightarrow k}$$

The empty boolean condition is a valid guard that is always evaluated as true.

9. **Property 9. Sequence control operator:** the sequence control operator is given by the simple interconnection of two or more references to diagrams in  $\mathcal{R}_{SD}$  and  $\mathcal{R}_{AD}$ . We formally define a sequence path as follows:

$$\begin{aligned} \text{path} = & (n_1, l_{n_1, n_2}), \{(n_i, l_{n_i, n_{i+1}})_{i:2 \rightarrow m-2}^+\}, (n_{m-1}, l_{n_{m-1}, n_m}, n_m) \\ & l \in L, n \in \{R_{SD}, R_{AD}\} \end{aligned}$$

10. **Property 10. Iteration control operator:** the iteration control operator (i.e., a for-loop) has one incoming and two outgoing edges. These two outgoing edges are connected to the body of the loop and to the exit branch. These two branches both constitute a path. A condition is composed of a set of 3 clauses, as in standard for-loops: an initialization, a stop condition and an increment.

$$\begin{aligned} \text{iteration} = & \text{condition, } \{\text{branch}_{exit}, \text{branch}_{body}\} \\ \text{branch}_{exit}, \text{branch}_{body} = & \text{path} \\ \text{condition} = & \text{initialization; stop\_condition; increment} \end{aligned}$$

11. **Property 11. No recursion:** recursion is not allowed for any outgoing branch of any control operator  $c \in \mathcal{C}_{op}$ .
12. **Property 12. Start and final nodes:** an Activity Diagram is allowed to contain one and only one start node. It must contain at least one final node. Moreover, the following links are not valid connections between nodes:

$$\{l_{n,start}, l_{final,n}\}, \text{ with } l \in L, n \in N$$

13. **Property 13. Mapping of instances of type storage:** An instance  $i \in \mathcal{I}_{SD}$  of type  $i \in controller$  cannot be mapped onto a memory block that describes a cache memory in a platform model. In DIPLODOCUS, the effects of caching are modeled by the *cache-miss ratio* parameter of a generic CPU block [7,8].

## Appendix 1.B TTool/DIPLODOCUS' simulation semantics

The text in this appendix is an excerpt of [4] that informally describes and classifies the simulation semantics into the three categories: Functionality, Platform and Mapping. Assumptions were made for three reasons: first, some of them stem from abstractions inherent to the DIPLODOCUS model of computation which is especially tailored to performance aspects. In the following, these assumptions are marked with (Perf). Other assumptions (denoted by (Sim)) are either technical and simply facilitated the implementation of the simulator or are of descriptive nature. A relaxation of these assumptions is envisaged in the future. Assumptions indicated by (Sys) are normally driven by particularities of the system to be modeled. (Sys) assumptions were obtained from the insights gained in various projects and case studies.

### 1.B.1 Functionality

Concerning the functionality of the system under design (application and communication models), the following assumptions were made:

- (Perf): A transaction is considered to be monolithic in the sense that the partial order of actions within a transaction is not resolved. Branch prediction penalties are spread uniformly across a transaction (cf. section 5.4.2), and so it is not intended to specify their start and finish time. This assumption comes with the positive effect of reducing indeterminism and augmenting simulation performance while being justifiable at the given level of abstraction. Performance figures primarily depend on which control flow path in a DIPLODOCUS task is taken. The respective guards (of Choice commands) are mostly time invariant, as long as they do not refer to a value obtained from a Notified Command. (Recall: This command returns the number of events stored in a FIFO). By assuming a scarce use of Notified Commands, the overall workload imposed on the hardware architecture is largely independent of the timing. Therefore, given the inherent inaccuracy of high level models, the partial order within transactions is assumed to play a marginal role for performance measurements.
- (Perf): As a rule of thumb, control flow related commands do not advance simulation time whereas commands triggering transactions indeed do. The following commands are executed in zero time: Action, Choice, For Loops, Sequence, Random Sequence, Random Number. Other commands let time elapse: Write Channel, Send Event, Send Request, Read Channel, Wait Event, Notified Event, Select Event, ExecI, ExecC, Delay. The assumption requires the computational complexity and data transfers associated to control flow structures being neglectable as compared to those modeled with Write, Read and Exec commands.
- (Sys): In the simulation framework, synchronization is expected to have an impact on system performance. The cost of Send Event/Wait Event commands is provided as a parameter to the simulator. In case events are mapped onto a bus the cost is expressed in terms of bytes, otherwise it is considered as an ExecI unit. However, it could also be argued that the performance impact of data transfers for synchronization is neglectable with respect to the rest of the application. This heavily depends on the average length of computations and data transfers of the system to be modeled. The simulator is flexible enough to be easily adjusted to the different circumstances.
- (Sim): The application model embraces indeterministic commands such as Random, Random Sequence, Random Choice, ExecIInterval, ExecCInterval, DelayInterval. During simulation, indeterminism is simply resolved by means of a random number generator, whereas formal techniques and the exhaustive simulation explore all valuations of the particular random variable. This complies to prevalent model checking practices.
- (Sim): So far, Select Event commands are defined to be deterministic. Events are checked in the order in which they were connected in the graphical model.

These assumptions are expressed in terms of the operators that compose the activity diagram of a task in the application model. In fact, from the viewpoint of simulation, a Communication Pattern is translated into a representation that is equivalent to a set of interconnected tasks as in an application model. See [18] for more details about the transformation rules of a CP into an equivalent task representation.

### 1.B.2 Platform

In the simulator, the impact of caches, memories, CPU power saving strategies and operating systems are modeled in a rudimentary way. To increase the accuracy of simulation results, the proposed models should be refined and calibrated according to the used cache hierarchy, memory technology, power manager and operating system. In the field of simulation, the purpose of this work is to propose an efficient simulation strategy tailored to the properties of DIPLODOCUS models. The simulator has been designed with future enhancements in mind, which means that it

could easily accommodate a more sophisticated cache or energy manager model. The issue of adequately representing data and instruction caches has been investigated in the scope of a dissertation [10] simultaneously to this work. Moreover, there is an ongoing dissertation which addresses the refinement of the existing energy consumption models and their integration into the simulator.

Three deterministic penalties may be imposed on Exec operations:

- Power saving mode: If a CPU is not loaded with instructions for a given time, it enters an idle mode which reduces the energy consumption. When the CPU is in idle-mode, tasks requesting CPU processing time suffer from a constant wake-up delay.
- Context switch: The scheduler of the operating system may also degrade performance especially if task switches occur frequently. To account for the additional payload due to scheduling algorithms and context switching, a static task switching penalty is introduced. It delays a transaction on a CPU, if the previous transaction did not belong to the same task.
- Branch prediction: In pipelined CPU architectures, branch predictors attempt to guess which branch of a conditional instruction will be chosen. The purpose of the branch predictor is to avoid a pipeline-flush, implying that partially executed instructions have to be discarded. To account for this effect, the execution time of an Exec unit  $t_{exec} = \frac{cycles_{exec}}{f_{processor}}$  is multiplied with a correction factor  $t_{execi,branch} = t_{exec}(p_{miss} \times s_{pipeline} + 1 - p_{miss})$ , where  $0 \leq p_{miss} \leq 1$  denotes the branch miss probability of a conditional branching instruction and  $s_{pipeline}$  the number of stages to be flushed in case of a branch miss

Moreover, the following modeling assumptions have been made:

- (Sim): The master clock frequency is assumed to be a common integer multiple of the frequency of all clocked components (CPUs, buses, memories). However, this assumption could easily be eliminated.
- (Sys): When transferring data, the transmission speed is rather limited by the interconnect than by execution components. The duration of Exec transactions is therefore calculated as a function of CPU parameters, whereas the duration of data transfers solely depends on interconnect parameters. This assumption should hold for most of the modern chip architectures. Otherwise, it could easily be reconsidered by slightly adapting the simulator.
- (Sim): The throughput is determined by the weakest link, meaning the slowest bus or memory, and proportional to the transmitted data (for the time being, no static offsets are added for memory accesses, but this decision could easily be reconsidered).
- (Perf): Each interconnect component on a route may delay a transaction, thus recalculate the transmission time, or add a static access time.
- (Sim): Buses may be endowed with several independent communication channels which can be used simultaneously. This extends the scope of the model to other interconnect architectures like crossbars enabling multiple simultaneous connections.
- (Sim): The communication model bases on the circuit switching paradigm; for the time being packet switching cannot be represented. During the entire data transmissions on more than one bus, at least one communication channel on all involved buses has to be available and is reserved. This could be prevented by making bridges active components, that receive data on one bus and create a send transaction on another bus. Reservation is accomplished starting from the CPU towards the memory element in a causal fashion.
- (Sim): Deadlocks are not resolved in case different CPUs mutually try to reserve buses already reserved by the other one.
- (Sim): A memory has as many ports as it has connections to buses. In case a bus is equipped with several channels, all channels are assumed to have a separate memory port.
- (Perf): Hardware accelerators are modeled as CPUs with all penalties disabled. One CPU should be foreseen per task to avoid the implications of a scheduling policy. Indeed, an operation in DIPLODOCUS is only characterized by its complexity and therefore there is no fundamental difference in whether the operation is executed on a CPU or on a dedicated hardware component.
- (Sim): DMAs are represented with dedicated CPUs running tasks which accomplish the data transfer.

### 1.B.3 Mapping

Concerning the mapping, the following assumptions were made:

- (Sys): The amount of data carried by events may or may not be neglectable with respect to data transfers expressed with channels. Therefore, the simulation environment leaves the decision to the user whether events are mapped onto buses.

- (Sim): Data associated to an abstract channel is assumed to be located at one single physical position in the system. For that reason, a channel is implicitly mapped onto n buses, n - 1 bridges and 0 or 1 memory. This assumption is debatable given that embedded systems nowadays comprise a heterogeneous memory architecture (volatile and non-volatile memory of different techniques). This assumption is relaxed in [10].
- (Sim): If a channel is mapped onto a memory, the route connecting sending CPU, memory and receiving CPU may be inferred by the code generator. The channel does not have to be explicitly mapped onto buses or bridges. In case there exist several routes, the intended route should be marked with at least one mapping artifact. Otherwise the result is undefined as it depends on implementation internals of the code generator.
- (Sim): If a channel is mapped onto buses, read and write operations normally involve transactions on those buses. However, if a channel is not mapped onto any memory, it means that the data is buffered somewhere within the bus master of the receiver. In this case, a read transaction is conveyed on the bus whereas the write transaction does.

## References

1. Schmidt, D.C.: Model-Driven Engineering. *IEEE Computer* 39(2), (2006)
2. TTool, <http://ttool.telecom-paristech.fr>
3. Knorreck, D. and Apvrille, L. and Pacalet, R.: Fast Simulation Techniques for Design Space Exploration. International Conference TOOLS, pp. 308-327, (2009)
4. Knorreck, D.: UML-Based Design Space Exploration, Fast Simulation and Static Analysis. PhD dissertation, Telecom ParisTech, (2011)
5. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.: A Methodology to Design Programmable Embedded Systems - The Y-chart Approach. In: *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, pp. 18-37 (2002)
6. IEEE 802.15.4 group: IEEE 802.15 Wireless Personal Area Networks (WPAN) Task Group 4 (TG4). <http://www.ieee802.org/15/pub/TG4.html>, (2003)
7. Apvrille, L. and Muhammad, W. and Ameur-Boulifa, R. and Coudert, S. and Pacalet, R.: A UML-based Environment for System Design Space Exploration. In: *Electronics, Circuits and Systems (ICECS)*, pp. 1271-1275, (2006)
8. Apvrille, L.: TTool for DIPLODOCUS: An Environment for Design Space Exploration. In: NOTERE, pp. 28:1-28:4, (2008)
9. Waseem, M. and Apvrille, L. and Ameur-Boulifa, R. and Coudert, S. and Pacalet, R.: Abstract Application Modeling for System Design Space Exploration. In: *EUROMICRO DSD*, pp. 331-337, (2006)
10. Jaber, C.: High-Level SoC modeling and performance estimation applied to a multi-core implementation of a LTE enodeb physical layer. PhD dissertation, Telecom ParisTech, (2011)
11. R.M. Koteng: Evaluation of SDR-implementation of IEEE 802.15.4 Physical Layer, Master of Science thesis, Norwegian University of Science and Technology (NTNU), (2006)
12. Zimmermann, H.: OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. In: *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425-423, (1980)
13. Ousterhout, J.: Why threads are a bad idea (for most purposes). Available at: <https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf>, (1996)
14. Dabek, F. and Zeldovich, N. and Kaashoek, F. and Mazières, D. and Morris, R.: Event-driven Programming for Robust Software. In: 10th Workshop on ACM SIGOPS European Workshop, pp. 186-189, (2002)
15. Lee, E.A.: The Problem with Threads. Available at: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html>, EECS Department, UC Berkeley, (2006)
16. Gonzalez-Pina, J.M.: Application Modeling and Software Architectures for the Software Defined Radio. PhD Dissertation, Telecom ParisTech (2013)
17. Muhammad, N.u.-I., Rasheed, R., Pacalet, R., Knopp, R., Khalfallah, K.: Flexible Baseband Architectures for Future Wireless Systems. In: *EUROMICRO Digital System Design*, pp. 39-46. (2008)
18. Enrici, A.: Model Driven Engineering of Parallel and Distributed Systems: the  $\Psi$ -chart Approach. PhD dissertation, Telecom ParisTech (2015)
19. Mueller, W. and Ruf, J. and Hoffmann, D. and Gerlach, J. and Kropf, T. and Rosenstiehl, W.: The simulation semantics of SystemC. In: DATE, pp. 64-70, (2001)
20. GTKWave. Downloadable at <http://gtkwave.sourceforge.net/>
21. Graphviz - Graph Visualization Software. Downloadable at <http://www.graphviz.org/>
22. INRIA Rocquencourt and INRIA Grenoble Rhônes-Alpes: Construction and Analysis of Distributed Processes (CADP). Available at: [cadp.inria.fr](http://cadp.inria.fr)
23. Uppsala Universitet and Aalborg University: UPPAAL. Available at: [www.uppaal.org](http://www.uppaal.org)
24. INRIA Roquencourt, INRIA Grenoble Rhône Alpes, Laboratoire d'Informatique de Grenoble: Tutorials for CADP, LNT and LOTOS. Available at <http://cadp.inria.fr/tutorial/>
25. Latre, B., De Mil, P., Moerman, I., Dhoedt, B., Demeester, P. Throughput and Delay Analysis of Unslotted IEEE 802.15.4. In: *Journal of Networks*, vol. 1, n. 1, pp.20-28 (2006)
26. Waseem, M., Apvrille, L., Ameur-Boulifa, R., Coudert S., Pacalet, R. Abstract Application Modeling for System Design Space Exploration, *EUROMICRO DSD*, pp. 331-337 (2009)
27. Zedboard, <http://zedboard.org/>
28. Blanchet, B. Automatic Verification of Correspondences for Security Protocols. In: *Journal of Computer Security*, vol. 17, no. 4 (2009)