Integration of SystemC-AMS Simulation Platforms into TTool

Rodrigo Cortés Porto

▶ **To cite this version:**

Rodrigo Cortés Porto. Integration of SystemC-AMS Simulation Platforms into TTool. Computer Science [cs]. TU Kaiserslautern, 2018. English. NNT : . tel-02485710

HAL Id: tel-02485710

https://hal.sorbonne-universite.fr/tel-02485710

Submitted on 20 Feb 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

University of Kaiserslautern

Department of Electrical Engineering and Information Technology

Microelectronic Systems Design Research Group

Master Thesis

Integration of SystemC-AMS Simulation Platforms into TTool

Presented:	November 26, 2018
Author:	Rodrigo Cortés Porto (402425)
Research Group Chief:	Prof. Dr.-Ing. N. Wehn
Tutors:	Dr. D. Genius (Sorbonne University, LIP6)
	Prof. Dr. C. Grimm (TU Kaiserslautern, CPS)
	Dr.-Ing. M. Jung (TU Kaiserslautern, EMS)

Statement

I declare that this thesis was written solely by myself and exclusively with help of the cited resources.

Kaiserslautern, November 26, 2018

Rodrigo Cortés Porto

Acknowledgments

I would like to express my deepest gratitude to Dr. Daniela Genius, my supervisor at the *LIP6* (Laboratoire d'Informatique de Paris 6, France), for giving me the opportunity to work on the project "SystemC-AMS extensions for TTool", and providing me not only great technical guidance but also valuable support on my administrative issues before and during my whole stay in France.

I would like to thank the *CONACyT* (Consejo Nacional de Ciencia y Tecnología, México), for the economical support that they provided me which made it possible to carry out my Master's studies in Europe.

I would like to express my gratitude to Dr. Marie-Minerve Louërat, head of the System-On-Chip department at the LIP6, and Dr. François Pêcheux, professor at the LIP6, for their guidance and advise that helped to define the path of this work. I also would like to thank Dr. Liliana Andrade for the help provided on the CPN and SystemC-MDVP topics. I am thankful to Irina Lee, master student at the Sorbonne University, for her hard and excellent work on the SystemC-AMS graphical interface for TTool and for her proactivity during her traineeship working on this project.

I would like to thank my professors at the TU Kaiserslautern, Prof. Christoph Grimm, head of the Design of Cyber-Physical Systems chair, for his support in this project, agreeing to be my supervisor from the TU Kaiserslautern side and being the principal contact between the LIP6 and the TU Kaiserslautern; Prof. Norbert Wehn, head of the Microelectronic Systems Design research group, and Dr. Christian Weis, professor at the Microelectronic Systems Design research group, for their support in accepting this project to be developed as part of my Master's thesis; Dr. Matthias Jung, from the Fraunhofer *IESE* (Institute for Experimental Software Engineering), for his support in being my co-supervisor from the Electrical and Computer Engineering department at the TU Kaiserslautern side; Prof. Wolfgang Kunz, head of the Electronic Design Automation chair, for his great support and guidance during my Master's studies.

I would like to thank my professors at the Tecnológico de Monterrey, Mexico City Campus, Dr. Katya Eugenia Romo-Medrano Mora, Dr. Alfredo Victor Mantilla Caeiros and Dr. Edgar Omar López Caudana, for their great support and recommendations, that opened me the doors to come to Germany to start my Master's studies.

I would like to acknowledge the great administrative support and advise from Aurore Marcos and Nadia Anbajagan here at the LIP6; and from Arthur Harutyunyan, from the International School for Graduate Studies at the TU Kaiserslautern.

I would like to thank Dr. Hassan Aboushady and Dr. Haralampos Stratigopoulos for their advise and friendship during my work at the LIP6. I would also like to express my gratitude to my office mates and friends at the LIP6, and to all my friends at the TU Kaiserslautern, for their daily friendship and great work environment that they built.

Finally, I would like to express all my gratitude to my parents and the rest of my family, for all the support and love that they have given me during my whole studies.

Abstract

Embedded systems consist of the integration of software and hardware components, where the latest are increasingly of a heterogeneous nature, composed of digital and analog integrated circuits, sensors and actuators. A heterogeneous modeling, including analog/mixed signal and radio frequency components, is a requirement for the design and development of many embedded systems applications.

The integration of analog components into a high-level modeling and virtual prototyping tool, named TTool, is presented here. The idea is to integrate the analog components as targets into a digital model of an MPSoC that runs software under an operating system. The resulting virtual prototype can be simulated at cycle-accurate/bit-accurate level.

The development of this thesis is based on the results of a previous work, in which the graphical interface of TTool has been augmented to allow the creation of analog/mixed signal models without including any digital component, and which could already generate operational SystemC-AMS code of the models.

The integration tasks are divided in two stages: during the first stage, the integration between the analog and digital components has been implemented independently of TTool, but using the same digital components that TTool uses without running any operating system nor software. In the second stage, the integration of the analog and digital components into TTool has been performed. During this stage, the validation of the analog models at the design level before the generation of the virtual prototype has been implemented.

Two case studies are presented to demonstrate the outcome of this work. First, a rover system is modeled, consisting of an MPSoC digital platform and the models of two analog sensors. A virtual prototype including embedded software is generated and simulated based on this model. Second, a model of a vibration sensor is created and its virtual prototype is generated and simulated. The results of the simulation of this model, along with the validation at the design level are compared against the outcomes of the model being simulated directly using SystemC-AMS and SystemC-MDVP. Several other models are presented to expose particular problems that may arise during the integration of analog and digital models, specially when dealing with the interaction of Discrete Event and Timed Data Flow Models of Computation. These models are also used to compare the behavior of the results of this work with the behavior from the models created directly using SystemC-AMS and SystemC-MDVP.

Zusammenfassung

Eingebettete Systeme bestehen aus der Integration von Software- und Hardwarekomponenten, von denen die letzten zunehmend heterogener Natur sind, bestehend aus digitalen und analogen integrierten Schaltkreisen, Sensoren und Aktoren. Eine heterogene Modellierung einschließlich Analog- / Mixed-Signal- und Radiofrequenzkomponenten ist eine Voraussetzung für das Design und die Entwicklung vieler Eingebettete Systeme-Anwendungen.

In der vorliegenden Arbeit wird die Integration analoger Komponenten in ein High-Level-Modellierungs- und virtuelles Prototyping-Tool namens TTool vorgestellt. Die Idee besteht darin, die analogen Komponenten als Ziele in ein digitales Modell eines MPSoC zu integrieren, auf dem Software unter einem Betriebssystem ausgeführt wird. Der resultierende virtuelle Prototyp kann auf cycle-accurate/bit-accurate Niveau simuliert werden.

Die Entwicklung dieser Arbeit basiert auf den Ergebnissen einer früheren Arbeit, in der die grafische Benutzeroberfläche von TTool erweitert wurde, um die Erstellung von Analog- / Mixed-Signal-Modellen ohne jegliche digitale Komponente zu ermöglichen, und die bereits operatives SystemC-AMS-Code der Modelle generieren konnte.

Die Integrationsaufgaben sind in zwei Stufen unterteilt: Während der ersten Stufe wurde die Integration zwischen analogen und digitalen Komponenten unabhängig von TTool implementiert, wobei jedoch dieselben digitalen Komponenten verwendet wurden, die TTool verwendet, ohne dass eine Software und das zu deren Ausführung benötigte Betriebssystem verwendet wird. In der zweiten Stufe wurden die analogen und digitalen Komponenten in TTool integriert. In dieser Stufe wurde die Validierung der analogen Modelle auf Designebene vor der Generierung des virtuellen Prototyps implementiert.

Es werden zwei Fallstudien vorgestellt, um die Ergebnisse dieser Arbeit zu demonstrieren. Zunächst wird ein Rover-System modelliert, das aus einer digitalen MPSoC-Plattform und den Modellen zweier analoger Sensoren besteht. Basierend auf diesem Modell wird ein virtueller Prototyp mit eingebetteter Software generiert und simuliert. Als zweites wird ein Modell eines Vibrationssensors erstellt und sein virtueller Prototyp erstellt und simuliert. Die Ergebnisse der Simulation dieses Modells sowie die Validierung auf Entwurfsebene werden mit den Ergebnissen des Modells verglichen, das direkt mit SystemC-AMS und SystemC-MDVP simuliert wird. Verschiedene andere Modelle werden vorgestellt, um besondere Probleme aufzuzeigen, die bei der Integration von analogen und digitalen Modellen auftreten können, insbesondere wenn es um die Interaktion zwischen Discrete-Event- und Timed-Data-Flow Berechnungsmodellen geht. Diese Modelle werden auch verwendet, um das Verhalten der Ergebnisse dieser Arbeit mit dem Verhalten der Modelle zu vergleichen, die direkt mit SystemC-AMS und SystemC-MDVP erstellt wurden.

Contents

1. Introduction

1.1. Context

Due to the high complexity of today's embedded systems, model-driven development techniques are a current practice for the design and development of embedded software. These techniques make use of high level models to create the specification of the software, and then perform transformations of the models to generate the source code [1, p. 10].

These approaches however are limited to the digital part of the system. Embedded systems are often of a heterogeneous nature, composed of digital and analog–*AMS* (analog/mixed signal) and *RF* (radio frequency)–components. For this purpose, many heterogeneous embedded systems require a modeling that includes models of their AMS and RF components. Moreover, the possibility to run software on the digital part is required.

TTool [2] is a design tool for safe and secure digital embedded systems, developed at Telecom ParisTech and containing extensive verification capabilities. It has been extended by LIP6 with the possibility to generate cycle-bit accurate virtual prototypes for a full-system simulation—including an *OS* (Operating System), embedded *SW* (software) and a model of *HW* (hardware) [3].

The present work is part of the LIP6 project "SystemC-AMS extensions for TTool", which aims at the addition of models for analog hardware components, with the aim of obtaining a platform that can run software.

Abstractions of the analog hardware components are necessary for the generation of the SystemC/SystemC-AMS virtual prototype of the model. The addition of analog hardware components into TTool for the generation and simulation of mixed analog and digital virtual *MPSoC* (multi-processors system on chip) prototypes consists of three phases:

1. The first phase, which was developed as part of [4], consists of generating fully functional SystemC-AMS topcells in TTool, by limiting to platforms without any software part and without any preexisting SoCLib[5] component. A graphical interface has already been augmented, offering the possibility to describe SystemC-AMS modules within Timed Data Flow clusters.

2. In the second phase, the intention is to combine the SystemC-AMS part with a digital SoC platform modeled with SoCLib components. First, we will provide assembler instructions directly to the memory blocks of the SoC platform. Later, C code will be cross-compiled to the specific CPU architecture of the model and loaded to the memory blocks of the SoC platform. In this phase no OS will be used.

3. In the third phase, the SystemC-AMS modules created in TTool will be integrated with the digital hardware components of TTool (SoCLib components). We intend to run larger scope software, generated directly from the Software Design Diagrams of TTool and requiring an OS and a linker script.

1.2. Objective

The focus of this work is on the second and third phases of the LIP6 project "SystemC-AMS extensions for TTool" described above. The main objective is to extend the functionality of TTool to generate virtual prototypes of heterogeneous embedded systems composed of digital hardware (MPSoC) and analog hardware, that can run embedded software using an OS (MutekH [6]). This will be done by integrating the SystemC-AMS components with a digital MPSoC platform based on SoCLib components including embedded software and the MutekH OS; and by implementing a solution to the time synchronization issues that may occur between the *DE* (Discrete Event) and *TDF* (Timed Data Flow) *MoC* (Models of Computation).

1.3. Thesis organization

In Chapter 1, the context and objective of this thesis is defined. The organization of the rest of the document is presented below.

In Chapter 2, related work to this thesis is presented. The AMS extensions for SystemC are described. The SoCLib library is introduced and its main features are pointed out. An approach developed as part of the SystemC-*MDVP* (Multi-Disciplinary Virtual Prototyping) simulator that deals with the problem of time synchronization between the TDF and DE MoCs is explained. An overview of TTool and the SystemC-AMS graphical interface and platform generation developed in phase 1 is shown.

In Chapter 3, the development of phase 2 of the LIP6 project "SystemC-AMS extensions for TTool" is described, which consists in the integration of SystemC-AMS and SoCLib components without the use of an OS. Two different steps are carried out: first by using a pedagogic SoCLib SystemC model of a one memory mono-core SoC in which assembler instructions were inserted; and later using a SoCLib model of a similar one memory mono-core SoC in which C code was programmed. It is explained as well the need of creating a generic adapter module for connecting the analog and digital components of the models.

In Chapter 4, the time synchronization issues between the TDF and DE MoCs interactions are described more deeply. A description of the method proposed in [7] is presented. Finally, a solution for these synchronization issues within the context of this thesis is proposed.

In Chapter 5, the development of phase 3 is explained, which integrates the work done from phase 1 and 2 by using TTool's SoCLib and SystemC-AMS modules along with the MutekH OS. The implementation into TTool of the solution for the synchronization issues between the TDF and the DE MoCs presented in Chapter 4 is described.

In Chapter 6, simulation results of different test models are presented. A model that initially stems from a purely digital model in TTool, a rover system meant to assist rescuers to find victims in debris, is used as a case study. A vibration sensor model from the H-Inception project [8] has been modeled and simulated in TTool and is used to compare the results obtained by the SystemC-MDVP simulator as part of the work of [7]. Other SystemC-AMS models have been created in TTool, in SystemC-MDVP and

in SystemC to show how the synchronization issues between TDF and DE MoCs are handled in each simulator and compare the results.

Finally in Chapter 7, the conclusions and perspectives of this work are presented.

2. Related work

2.1. SystemC and SystemC-AMS

SystemC [9] is a C++ class library which supports the modeling of embedded systems and represents their hardware components as modules; these modules are connected using ports. The scheduling and synchronization of concurrent processes uses events and sensitivity of the ports. SystemC advances simulation time, separates computations (processes) from communication (channels); it supports hardware oriented data types [10, p. 562].

The SystemC simulation kernel is based on the DE MoC, which describes systems based on temporal sequences of a countable number of events [11]. The processes of a DE MoC describe the functional behavior of the system. The execution of these processes is triggered by events (inputs are sensitive to these events) or by the passing of time (after a "wait" of certain time). The processes themselves can generate new events. All events are sorted with respect to time stamps, and inserted into an event queue. A scheduler determines which processes can be executed next, based on the events from the event queue and the passing of time. Since the schedule is computed during run time, it is known as dynamic scheduling.

The SystemC-AMS extensions increase the capabilities of the SystemC library to allow the design and simulation of AMS and signal processing systems along with digital hardware systems [12, p. 169]. SystemC-AMS can be used for creating system-level models which can be used for different use cases such as executable specification, virtual prototyping, architectural exploration and validation of the system [13, p. 1 f].

SystemC-AMS extensions are implemented as a C++ class library. As shown in Figure 2.1, they are built on top of the SystemC standard. SystemC-AMS defines different modeling formalisms implemented by using different MoCs, to support AMS behavioral modeling. The MoCs that are currently in SystemC-AMS are *TDF* (Timed Data Flow), *LSF* (Linear Signal Flow) and *ELN* (Electrical Linear Networks).

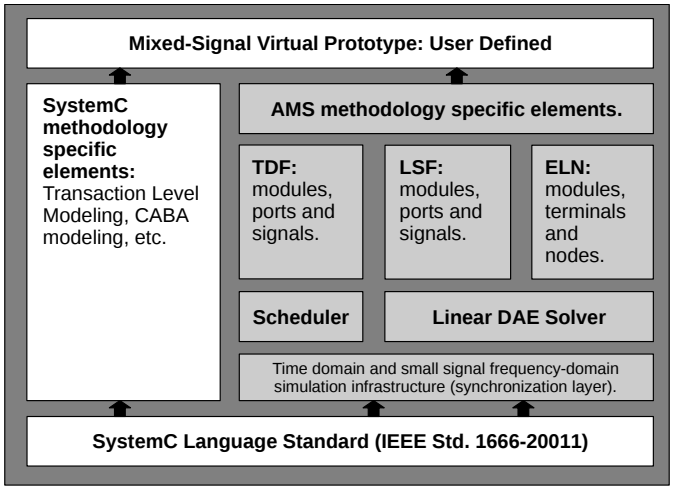


Figure 2.1: SystemC-AMS extensions. (Adapted from [12].)

The focus of this work is on the SystemC-AMS TDF MoC. It is based on the *SDF* (Synchronous Data Flow) [14] modeling formalism with the difference that TDF is a discrete-time model which considers the data as signals sampled in time, while SDF is an untimed MoC. As explained in [13, p. 7], the TDF model consists of nodes (modules) that represent processes; and arcs (signals) that represent data paths. TDF modules use ports as an interface to connect to other modules through signals. To connect to other TDF modules, TDF ports are used. To connect to DE modules, TDF converter ports are used. A set of connected TDF modules form a directed graph called TDF cluster. Figure 2.2 shows a TDF cluster, where the DE modules are represented as white blocks; TDF modules as gray blocks; TDF ports as black squares; TDF converter ports as black and white squares; DE ports as white squares; and TDF signals as arrows. The TDF modules of a cluster have attributes which are described below.

1. Module Timestep (**Tm**): denotes the period in which the module will be activated. One module will only be activated if there are enough samples available at its input ports.

2. Rate (**R**): Each module will read or write a fixed number of data samples each time it is activated. This number is annotated to the ports and it is known as the port rate.

3. Port Timestep (**Tp**): it is the period at which each port of a module will be activated. It also denotes the time interval between two samples that are being read or written.

4. Delay (**D**): A delay can be assigned to a port. As its name suggests, this attribute will make the port to hold up a number of samples each time it is activated, and read or write them in the following activation of the port.

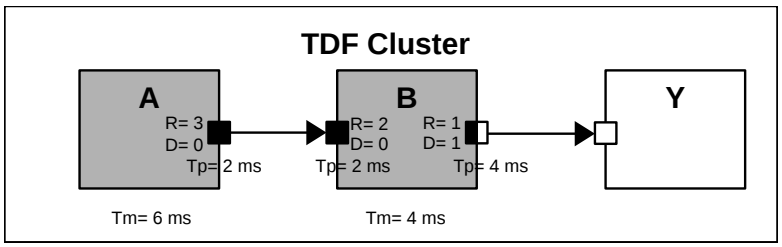


Figure 2.2: TDF Cluster

Compared to DE models, TDF models have the advantage that they can execute without the overhead of dynamic scheduling. Instead, they define a static schedule which is calculated before the simulation starts, accelerating the simulation's run time. Once the static schedule is computed, the TDF modules are executed according to this schedule.

2.2. SoCLib

As specified in [5], SoCLib is an open SystemC library for the creation of virtual prototypes of MPSoCs. It consists of simulation DE models for virtual components which can be simulated with the SystemC simulation environment. SoCLib components are implemented in two ways, based on the type of simulation that is required:

1. *CABA* (cycle-accurate bit-accurate) for very accurate but slower simulations [15]. TTool mainly focuses on the use of these components.

2. *TLM* (Transaction Level Modeling) [16] for faster simulations with some loss of accuracy.

SoCLib components are interconnected based on the *VCI* (Virtual Component Interface) [17] protocol. This makes the components easily inter-operable and facilitates the integration of new components into the platform. As noted in [17], it is important to mention that the VCI is not a bus but an interface. Hence, it specifies a request-response protocol, a protocol for transfer of requests and responses, and the coding and contents of these requests and responses. This interface can be used to connect two components in a point-to-point connection. One of the components will act as an initiator, issuing requests; while the other will act as a target, responding to these requests. Also, the VCI can be used as an interface to a wrapper, in other words, as a connection to a bus. In this way, the components can be connected to any bus.

SoCLib makes use of this interface to define its own interconnect component models. These interconnect components can be of two types:

1. Physical interconnects, which implement interconnects for which an equivalent physical hardware exists.

2. Virtual interconnects, which implement the behavior of an interconnect without having an equivalent physical hardware, which makes simulations faster. TTool implements this type of interconnects which can be either a *VGMN* (virtual generic micro-network) or a *VGSB* (virtual generic system bus).

Many SoCLib components have a VCI interface. SoCLib implements special signals to use with the VCI protocol. By doing so, the SoCLib hardware components may connect to the interconnect components via their VCI interfaces and using this VCI signals.

2.3. SystemC-MDVP and timed-CPNs

The SystemC-MDVP simulator was developed at LIP6 as part of two previous works [7, 18]. This simulator targets the simulation of heterogeneous systems which can be modeled using different MoCs. Within the context of SystemC-MDVP a solution for the detection of the time synchronization issues between the DE and TDF MoCs during the elaboration phase of the SystemC model is proposed. The approach is based on an equivalent representation of the TDF clusters and the interactions with the DE domain using *timed-CPN* (timed-Coloured Petri Nets).

CPN (Coloured Petri Nets) is a graphical discrete-event modelling language that can be used to create models and analyze the properties of concurrent systems. It combines the capabilities of Petri Nets and those of a high level programming language. Finally, it provides the primitives for the definition of data types, for describing the manipulation of the data, and for creating compact models that can be parameterisable.[19, p. 3] As it is described below, CPN can be used to represent TDF modules and their connections. Finally, if timing information is required, it can be added to the CPN models. In this case they are known as timed-CPNs.[19, p. 231]

As mentioned before, the TDF MoC is based on the SDF modeling formalism. It is possible to transform SDF models into Petri Nets by means of a set of translating mechanisms [20]. Since an SDF model may be represented by using Petri Nets, a TDF model can make use of Petri Nets as well in order to perform pre-simulation analyses of the model that do not include interactions between the TDF and DE time domains. These pre-simulation analyses do not take into account the time notion handled by the TDF ports and modules of a TDF cluster, but they can be useful in order to obtain a static schedule of the model. To be able to represent the timing information handled by the TDF converter ports and their interactions with the DE domain, timed-CPNs have showed to be a good choice for this purpose since they facilitate the detection of time synchronization problems and the proposition of a possible solution to these issues.[7, p. 56 ff] As it is described in detail in [21] and [7, p. 60 ff], the construction of an equivalent representation of the TDF clusters and their interactions with the DE domain using timed-CPN requires a three steps process, which is summarized below.

1. The first step is to represent a TDF module as an equivalent CPN. Figure 2.3 shows this equivalent CPN representation where a *transition* (square) represents the execution action of the TDF module. This transition has a name $\mathbf{M{:}}q_{\mathbf{M}}$; a guard $[j_{\mathrm{M}} <> q_{\mathrm{M}}]$ that evaluates if the transition is enabled or not, where j_{M} is the current execution number of the module and q_{M} is the maximum execution number of the module; and a priority *LOW*. The current execution number j_{M} is stored in a *place* (ellipse) which has a name **Counter M**; a color set INT that defines the data type; and the initial tokens of the place $InitCount_{\mathrm{M}}$.

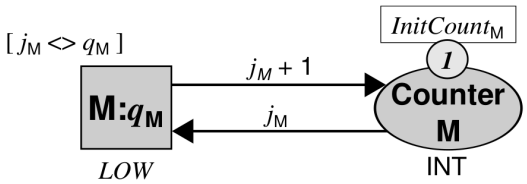


Figure 2.3: Equivalent CPN of a TDF module. (Reprinted from [7, p. 60] with permission.)

2. The second step consists on representing the TDF connections or signals as an equivalent CPN. Figure 2.4 shows the equivalent CPN representation of a TDF signal where the modules that are being connected are represented in a simple way by squares, and the signal is represented using a *place* (ellipse). The place has a name $\mathbf{S_N}$; a color set INT that defines the data type; and a multiset of delay tokens $DelayS_{\mathrm{N}}$ which are the addition of the delay attributes of the interconnected ports. The producer and transition modules are linked via directed arcs with functions that calculate the token identifiers produced or consumed when a module is fired.

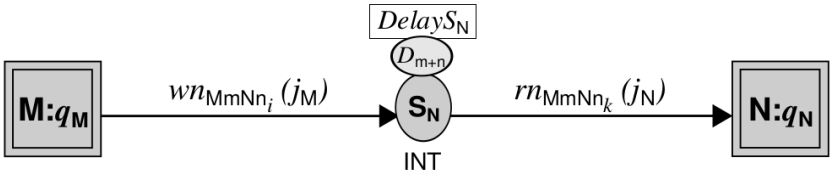


Figure 2.4: Equivalent CPN of a TDF signal. (Reprinted from [7, p. 62] with permission.)

3. The third step is to represent the TDF input and output converter ports as equivalent timed-CPNs. Each port is transformed into a series of transitions and places which control the read and write synchronization events between the TDF and DE MoC. This representation is explained in detail in [7].

Finally, it is shown in [21, 7] that by using an equivalent timed-CPN model for the DE-TDF model, pre-simulation analysis can be performed in order to detect any causality errors and if necessary, provide suggestions on the needed delay modifications that will avoid these time synchronization issues. Using this representation, the SystemC-MDVP simulator can can detect time synchronization issues between the TDF and the DE MoCs before the simulation phase starts, providing as an output to the user a list of Delay parameters that need to be added to the TDF converter ports in order to avoid causality problems.

2.4. TTool

TTool, developed mainly at Télécom ParisTech with LIP6 contribution on code generation and OS for MPSoC, is a UML and SysML based free and open-source software for model-based design and development of embedded systems at two different levels: partitioning (blue upper section of Figure 2.5) and embedded software design (blue lower section of Figure 2.5). According to TTool's philosophy, models are validated before any code is generated and the generated code is correct-by-construction.[2]

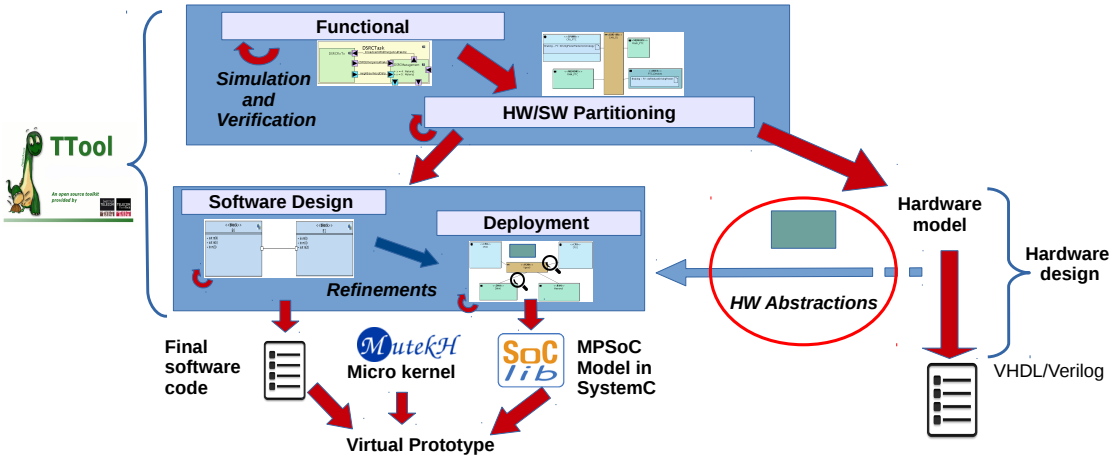


Figure 2.5: TTool's different levels for model-based design and development of embedded systems. (Adapted from [22]).

At the partitioning level, Functional and HW/SW Partitioning diagrams can be created in order to find the best software and hardware candidates that can execute the required functions of the system.

1. In the Functional Diagram, the functions of the system, which later will be assigned and implemented in software or hardware, can be modeled.

2. In the HW/SW Partitioning Diagram, the possible hardware architectures for the system can be modeled either as execution nodes (CPUs and HW Accelerators), communication nodes (buses) and storage nodes (memories) [2]. Finally in the

HW/SW Partitioning Diagram, the functions defined in the Functional Diagram are mapped to the hardware architecture and they can be allocated either to CPUs or HW Accelerators, while the communication between the functions can be assigned to specific communication and storage nodes.

At the embedded software design level, Software Design and Deployment diagrams can be created. They target the design of software as well as modeling and formal verification of embedded systems.

1. In the Software Design Diagrams, two types of diagrams can be created: Block Diagrams and State Machine Diagrams. With these two diagrams, the system software can be designed as communicating block diagrams and the behavior of each block can be implemented as state machines.

2. With the Deployment Diagrams (*DD*), each block representing the system software can be mapped to the chosen hardware architecture (CPUs or HW Accelerators) and their internal variables can be mapped to memories.

The Functional, HW/SW Partitioning and Software Design Diagrams are subject to formal verification and validation before the embedded software code can be generated.

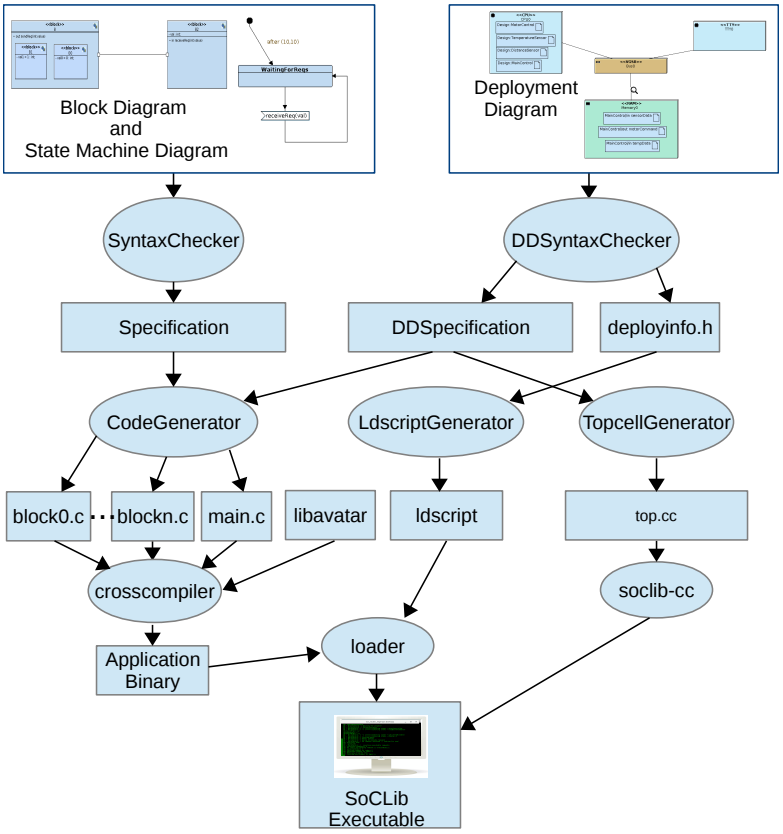


Figure 2.6: Toolchain for the software design level (Adapted from [3]).

In Figure 2.6, the Toolchain of the software design level is shown. From the software design level in the left, using a Block Diagram, the tasks of the components of a system can be represented by Blocks. Once a Block is created, a State Machine Diagram for

that Block can be created, which allows to design the software of the specific task. When the software design is complete, the SyntaxChecker will verify the correctness of the diagrams and will generate a Specification for the software. This Specification along with the DDSpecification is used by the CodeGenerator to generate one source code file per block and a `main.c` file. Using the `libavatar` libraries and the MutekH OS (a free portable OS for embedded systems with which several CPU architectures can be targeted [6]), the source code will be cross-compiled to the selected CPU architecture in order to generate an application binary.

In the Deployment Diagrams hardware abstractions are used as shown in Figure 2.5. In the work of [3], the abstraction of digital hardware components into the Deployment Diagram has been addressed and model transformations given, where hardware components can be transformed into SoCLib modules to generate a SystemC virtual prototype of the model and simulate it with a *CABA* (cycle-accurate bit-accurate) approach. Once the Deployment Diagram is complete, the DDSyntaxChecker shown in Figure 2.6, will verify that the Deployment Diagram is syntactically correct and generate a DDSpecification, used by the CodeGenerator and the `deployinfo.h` file. The `deployinfo.h` file contains information used by the LdscriptGenerator to generate the linker script (`ldscript`) which will be used by `Loader` utility of SoCLib later. The DDSpecification is also used by the TopcellGenerator to generate a SystemC topcell (`top.cc`) file, which includes all the required SoCLib modules for the model. This topcell can be compiled using the `soclib-cc` command to build a platform of the model or SoCLib Executable as it is shown in Figure 2.6. The topcell generates the mapping table for the different components of the model. It instantiates all the components from the model and creates the required signals to interconnect them. It creates the Net-List where all the interconnections of the components are defined. It also calls the `Loader` utility from SoCLib, which uses the `ldscript` to load the application binary into the RAM memory component of the model. Finally the SoCLib executable can be run to start the simulation of the virtual prototype.

2.5. SystemC-AMS graphical interface and platform generation in TTool

During the first phase of the project "SystemC-AMS extensions for TTool" developed as part of [4], the graphical interface of TTool was augmented, offering the possibility to create SystemC-AMS TDF models which consist of TDF clusters including TDF and DE blocks. As it shown in Figure 2.7, a new SystemC-AMS panel was added. In this panel, a new SystemC-AMS Component Diagram can be created where TDF models can be designed.

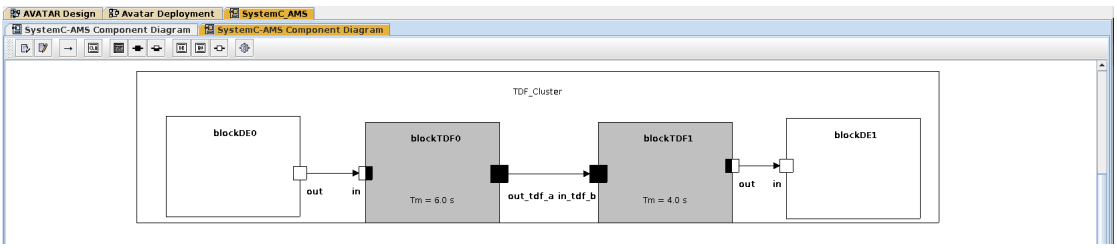
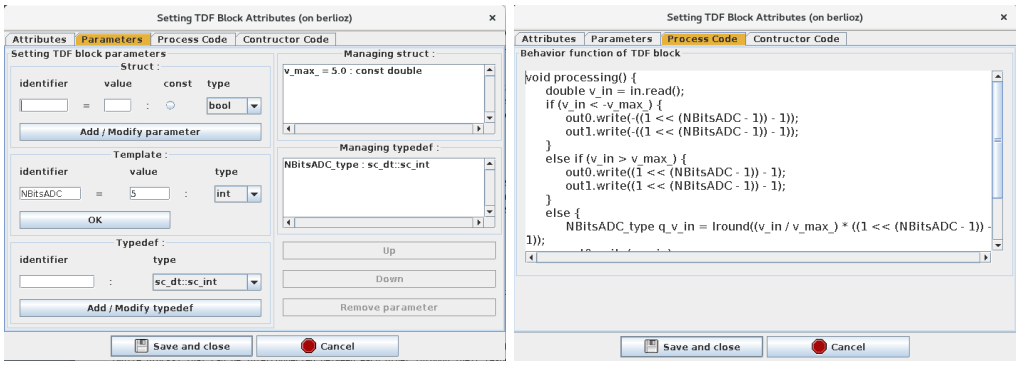


Figure 2.7: TTool's augmented graphical interface with SystemC-AMS Component Diagram.

A model consists of a TDF cluster which can contain TDF modules (gray blocks) and DE modules (white blocks) that can be interconnected with each other through their respective ports and signals. Several TDF clusters can be designed each in its own panel, in such way that they will be seen by the SoCLib initiator components (CPUs) as individual target components, when the integration of SystemC-AMS with SoCLib components is implemented.

To connect TDF modules, TDF ports (black ports) are used. To connect DE modules, DE ports (white ports) are used. In order to connect a TDF module with a DE module, converter ports (black and white ports) are used. The graphical interface offers a toolbar which allows to select the different components to build the model. The code for the TDF and DE modules functions should be provided directly as SystemC-AMS code through the graphical interface of TTool, as Figure 2.8b shows.

When a TDF module is created, it is possible to modify its attributes, parameters and provide manually the code that will define its behavior. The name and Timestep or Period (Tm) of a module can be defined. The period units can be selected as well between seconds, milliseconds, microseconds or nanoseconds. The parameters of a TDF module such as its internal variables or template parameters can be also set up, as it is shown in Figure 2.8a. Finally the code of the TDF module can be given manually. This code will define the SystemC-AMS `processing()` function, as it is depicted in Figure 2.8b.



(a) TDF module parameters. **(b)** TDF module process code.

Figure 2.8: Setting TDF module's attributes in TTool.

When a DE module is created, TTool's graphic interface allows to modify in a similar way its attributes, parameters and the code of the module's method.

Regarding the ports of the TDF and DE modules, their attributes can also be modified through the graphic interface of TTool. For converter ports and normal TDF ports, the name of the port can be introduced. The Period or Timestep of the port (Tp) can also be assigned along with its units (seconds, milliseconds, microseconds or nanoseconds). The Rate and Delay attributes can also be set up. The Type of the port can be chosen between `int`, `double` or `bool` for normal ports. Finally the Origin of the port should be set up, either to be an Input port or an Output port. For DE ports, the port can be added to the sensitivity list of the module by enabling the Sensitive field and selecting if the port will be sensitive to a positive or negative edge of the incoming signal or `null` for any incoming signal change.

As a side note, the graphical interface of TTool was also augmented to support the creation of SystemC-AMS models based on the ELN MoC. ELN integration is out of the focus of this thesis.

Once a TDF model is designed, the SystemC-AMS code can be generated at the click of a button through the TTool user interface. During this phase of the project, each block of the TDF model generates one header (`.h`) file which represents the SystemC-AMS or SystemC code of the TDF or DE block. The header file contains declarations of all the attributes of the block, such as its ports, timesteps, rates and delays; and the code for the functions that implement the behavior of the block.

Then, a source (`.cpp`) file is generated as a test-bench. It implements the `sc_main` function. This function defines the signals that connect the different blocks of the cluster. It instantiates all the TDF and DE modules from the TDF cluster. It also creates the interconnection Net-List of the modules by binding the module's ports to the signals. Finally, it starts the simulation for a default time of 100 milliseconds.

At last, a Makefile is generated. This Makefile can be used to compile the generated SystemC-AMS code and execute the simulation of the TDF model, without including any SoCLib component or embedded software.

3. Integration of SystemC-AMS and SoCLib components

3.1. Development of the GPIO2VCI SoCLib component

SoCLib is based on the shared memory paradigm, whose components are interconnected based on the VCI protocol. These components can be initiators which issue requests (e.g. CPUs) and targets that respond to these requests (e.g. RAM memory). The main idea for the integration of SystemC-AMS and SoCLib components in TTool is that the analog components will act as targets for the SoCLib initiator digital components (CPUs). In this sense, the generated topcell will be composed of SoCLib modules and the SystemC-AMS modules (TDF clusters). It is also important to mention that a TDF cluster may have DE modules within it, which are not part of the SoCLib library. This interaction is shown in Figure 3.1, where only the hierarchical composition of the modules is shown.

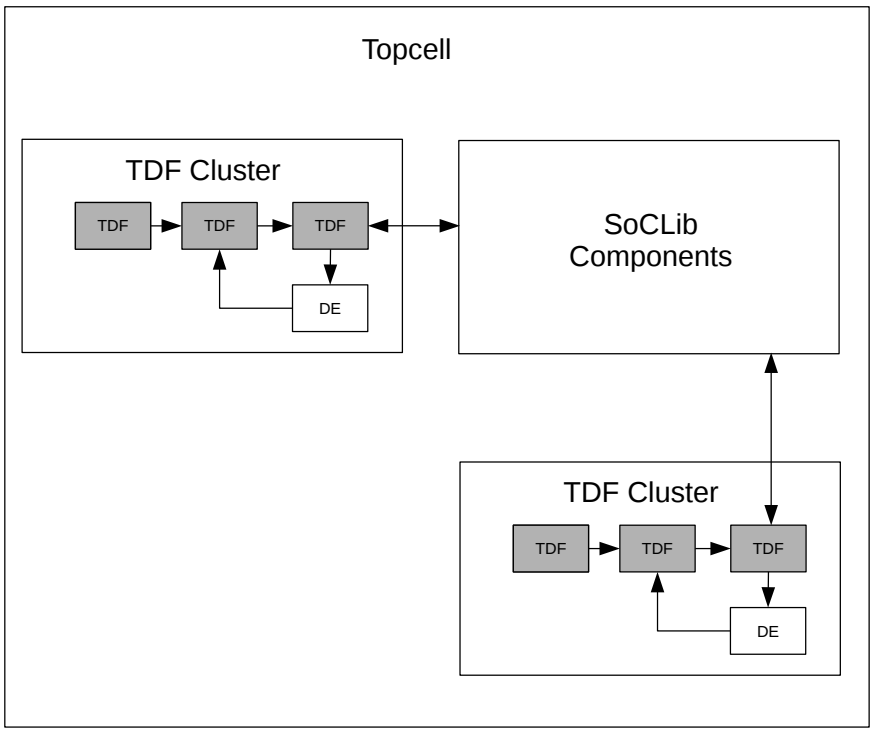


Figure 3.1: Integration of SoCLib and SystemC AMS components.

Due to the fact that SoCLib components are interconnected using the VCI protocol, the need of creating a generic adaptor module which can work as an interface between the SystemC-AMS modules and the the SoCLib interconnect components arose. This new component, which was modelled as a *GPIO* (general-purpose input/output) adaptor to VCI and called **GPIO2VCI**, was developed as a VCI target component following the modeling rules for writing CABA SystemC simulation models for SoCLib [15]. Figure 3.2 shows the model of this component and how it works as an interface between the SystemC-AMS modules (TDF_Module belonging to a TDF Cluster) and the SoCLib VCI interconnect component (VCI_Bus).

The previously mentioned modeling rules specify that the CABA components are built by one or several synchronous *FSM* (Finite State Machines) and have clearly identified internal registers. The FSM can be described by three types of functions. The `transition`

function, which is triggered once per cycle on the rising edge of the clock, will compute the next values of the registers, depending on their current values and the values of the input signals. The `genMoore` function, which is triggered once per cycle on the falling edge of the clock, computes the values of output signals that depend on the internal registers. The `genMealy` function, which is triggered once per cycle on the falling edge of the clock, computes the values of output signals that depend on the internal registers and the values of the input signals.

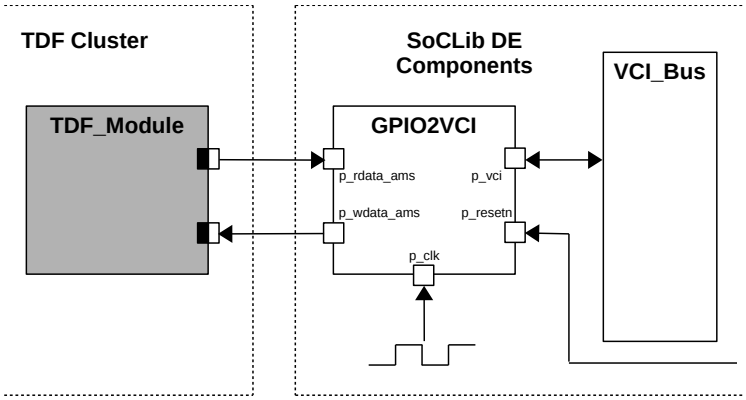


Figure 3.2: GPIO2VCI component.

The GPIO2VCI component's interface definition shown in Listing 3.1, contains the ports that a typical VCI target component has. These ports are an input clock port `p_clk`, a negative-edge reset port `p_resetn` and a VCI target port `p_vci`. It also has two ports used for communication with the SystemC-AMS modules. These ports are an input port `p_rdata_ams`, which is used to read data coming from the SystemC-AMS modules; and an output port `p_wdata_ams`, which is used to write data to the SystemC-AMS modules. It is important to mention that the data type `vci_param::data_t` of these two ports is internally defined as `sc_uint<32>`, so any SystemC-AMS module that is connected to the GPIO2VCI component through converter ports, should define this data type in these ports.

```
    sc_in<bool>                              p_clk;
    sc_in<bool>                              p_resetn;
    soclib::caba::VciTarget<vci_param>       p_vci;
    sc_in< typename vci_param::data_t >      p_rdata_ams;
    sc_out< typename vci_param::data_t >     p_wdata_ams;
```

Listing 3.1: GPIO2VCI ports definition.

```
#define sc_register sc_signal
...
    sc_register< typename vci_param::data_t > r_rdata_ams, r_wdata_ams;
    sc_register<int> r_fsm_state, r_buf_eop;
```

Listing 3.2: GPIO2VCI registers definition.

The internal registers definition shown in Listing 3.2, has two standard VCI target registers. These registers are the `r_fsm_state` register, that stores the next state of the FSM of the component; and the `r_buf_eop` register, which is used to mark an end of packet—i.e. end of a VCI transfer. It also has two new defined registers. The `r_rdata_ams`

register is used to store the data that was read from the SystemC-AMS components. The `r_wdata_ams` register is used to store the data that needs to be written to the SystemC-AMS components.

As defined in the modeling rules, the component implements two member functions `transition()` and `genMoore()`. The `transition()` function shown in Listing 3.3, will store into the internal registers of the component `r_rdata_ams` and `r_wdata_ams`, the values that need to be read from or written to the SystemC-AMS modules, depending on the VCI command `p_vci.cmdval` that is received. These values are read from the input ports of the component `p_rdata_ams` and `p_vci.wdata`.

```
tmpl(void)::transition() {
    if(p_resetn == false) {
        r_fsm_state = TARGET_IDLE;
    }
    else {
        switch( r_fsm_state ) {
        case TARGET_IDLE:
            if( p_vci.cmdval.read() ) {
                r_buf_eop = p_vci.eop.read();
                if ( p_vci.cmd.read() == vci_param::CMD_WRITE ) {
                    r_wdata_ams = p_vci.wdata.read();
                    r_fsm_state = TARGET_WRITE;
                }
                else {  //VCI_CMD_READ
                    r_rdata_ams = p_rdata_ams.read();
                    r_fsm_state = TARGET_READ;
                }
            }
            break;

        case TARGET_WRITE:
        case TARGET_READ:
            if( p_vci.rspack.read() ) {
                r_fsm_state = TARGET_IDLE;
            }
            break;
        }
    }
}
```

Listing 3.3: GPIO2VCI `transtion()` function implementation.

```
tmpl(void)::genMoore() {
    switch (r_fsm_state) {
    case TARGET_IDLE:
        p_vci.rspNop();
        break;
    case TARGET_WRITE:
        p_vci.rspWrite( r_buf_eop.read() );
        p_wdata_ams.write(r_wdata_ams);
        break;
    case TARGET_READ:
        p_vci.rspRead( r_buf_eop.read(), r_rdata_ams );
        break;
    }
    // We only accept commands in Idle state
    p_vci.cmdack = (r_fsm_state == TARGET_IDLE);
}
```

Listing 3.4: GPIO2VCI `genMoore()` function implementation.

The `genMoore()` function shown in Listing 3.4, will write the values on the output ports of the component depending on the state of the component's FSM. Besides the typical VCI output ports, it will write the values that need to be written to the SystemC-AMS modules through the `p_wdata_ams` output port.

The complete GPIO2VCI component definition and implementation code can be found in the Listings A.1 and A.2 from the Appendix.

3.2. Integration of SystemC-AMS modules with a pedagogic SoCLib model

The first step towards the integration of SystemC-AMS modules and SoCLib components without the use of an OS focuses on using an existing SystemC SoC model created for pedagogic purposes, taken from a Master's course in context of the H-Inception project [8]. This model consists of a 32 bits MiniMIPS processor *ISS* (Instruction Set Simulator), a simple 128 Bytes RAM memory, and a VCI local crossbar interconnect component. The components of the pedagogic SoCLib SoC are modeled in a similar fashion as the SoCLib components. The advantage of this model is that it does not require to install all the SoCLib libraries and utilities in order to compile and run. It also does not need any cross-compilers, since the software is written directly in assembler, which is processor specific, into the memory of the model. For these reasons, this model was chosen to be used as part of an initial approach for the integration with SystemC-AMS modules.

The simple Sine Source and the Sink SystemC-AMS modules, taken from a Master's course in context of the H-Inception project, were integrated with this one memory mono-core SystemC SoC model. As explained in Section 3.1, the GPIO2VCI component is used as an interface between the SystemC SoC model and the SystemC-AMS components. Figure 3.3 shows a simplified model of this SoC connected to the analog components via the GPIO2VCI component. It shows how the GPIO2VCI component is connected to the SystemC-AMS modules and to the VCI Interconnect, without including other internal ports like the clock or reset. The Sine Source block will generate a sine wave of a fixed frequency, which amplitude parameter is being controlled by the value that the GPIO2VCI component sends to this block. The values of the generated sine wave will be sent to the Sink block, which will print this value to the host computer terminal, and back to the GPIO2VCI component which will read this value and send it to the VCI Bus if required.

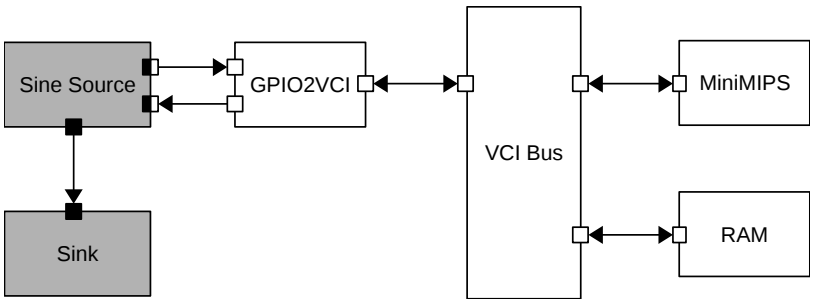


Figure 3.3: Model integrating SystemC-AMS and SystemC SoC components.

Since the SystemC SoC model uses a shared memory architecture, address space segmentation is used to assign segments of memory to the Hardware components via a mapping table. Listing 3.5 shows the declaration and construction of this mapping table, where three segments are added: the first two, `instructions` and `data`, are for the RAM instructions and data memory; and the third segment, `gpio2vci`, is for the GPIO2VCI component. In each segment the second field refers to the actual address assigned to the segment and the third field refers to the size of the segment. The fourth field indicates to which target each segment is mapped to. In this case, the `instructions` and `data` segments are mapped to target 0 (RAM component `t0`), while the `gpio2vci` segment is mapped to target 1 (GPIO2VCI component `t1`).

```
SOCLIB_MAPPING_TABLE  maptab (32, 1, intList(1), intList(1), 0x3<<29);
maptab.addSegment("instructions",     0x00000000, 0x40, maptab.ident(intList(0)),
SEGMENT_TYPE_CACHED);
maptab.addSegment("data",             0x00000040, 0x40, maptab.ident(intList(0)),
SEGMENT_TYPE_CACHED);
maptab.addSegment("gpio2vci",         0x80000000, 0x10, maptab.ident(intList(1)),
SEGMENT_TYPE_CACHED);
```

Listing 3.5: Mapping table for the SystemC SoC model.

For reading simplicity, the components have been instantiated with the names of the targets that they represent, as it is shown in Listing 3.6. It is important to note that the VCI Crossbar component `cb0` receives as templated parameters the number of initiators (1) and targets (2).

```
SOCLIB_VCI_ISS < VCI_PARAM > i0 ("i0");
SOCLIB_VCI_SIMPLERAM < VCI_PARAM > t0 ("t0");
GPIO2VCI < VCI_PARAM > t1 ("t1");
SOCLIB_VCI_LOCAL_CROSSBAR_SIMPLE<1, 2, VCI_PARAM> cb0("cb0", intList(), maptab);
SINE_SOURCE sine_1 ("sine_1");
SINK sink_1("sink_1");
```

Listing 3.6: Instantiation of the SystemC SoC model components.

Finally the Net-List is constructed by connecting the ports of the different components as it is shown in Listing 3.7. It can be noted that the GPIO2VCI component `t1` is connected to the SystemC-AMS component `sine_1` using the ports `p_rdata_ams` and `p_wdata_ams` (lines 12 and 13). It is important to note too that the VCI Crossbar component `cb0` specifies via its `TLOC_VCI` and `ILOC_VCI` ports to which initiator and target component it is connected (lines 17-19).

As explained before, the Sine Source component will read a value from the input converter port (`in`) that is connected to the GPIO2VCI component, and it will use that value as the amplitude parameter to generate a sine wave. The generated sine wave value will be converted to an integer and will be sent back to the GPIO2VCI component via its output converter port (`out`). In a similar way, the generated sine wave value will be sent to the Sink component via a normal output port(`tdf_out`). The behavior of the component is shown in the `processing()` function from Listing 3.8.

The Sink component will only read values from its input port (`in`) and print them to the console of the host machine, as shown in Listing 3.9.

```
1   i0.CLK(signal_clk);
2   i0.RESETN(signal_resetn);
3   i0.VCI_INITIATOR(ilink);
4
5   t0.CLK(signal_clk);
6   t0.RESETN(signal_resetn);
7   t0.VCI_TARGET(tlink0);
8
9   t1.p_clk(signal_clk);
10  t1.p_resetn(signal_resetn);
11  t1.p_vci(tlink1);
12  t1.p_rdata_ams(s_from_ams);
13  t1.p_wdata_ams(s_to_ams);
14
15  cb0.CLK(signal_clk);
16  cb0.RESETN(signal_resetn);
17  cb0.TLOC_VCI[0](ilink);
18  cb0.ILOC_VCI[0](tlink0);
19  cb0.ILOC_VCI[1](tlink1);
20
21  sine_1.in(s_to_ams);
22  sine_1.out(s_from_ams);
23  sine_1.tdf_out(sig_1);
24
25  sink_1.in(sig_1);
```

Listing 3.7: Net-List of the SystemC SoC model.

```
void processing() {
    // Get current time of the sample to be written to the out port.
    double t = out.get_time().to_seconds();
    // Calculate current value of the sine wave (amp = in.read(), f = 5 MHz)
    double x = in.read() * sin(2.0 * M_PI * 5000000.0 * t);
    // Write the value to the output ports
    out.write( (int) x);
    tdf_out.write(x);
}
```

Listing 3.8: Sine Source `processing()` function.

```
void processing() {
    using namespace std;
    cout << " " << this->name() << " @ " << this->get_time() << ": "
         << in.read() << endl;
}
```

Listing 3.9: Sink `processing()` function.

To test this first approach, assembler instructions need to be hard-coded to the memory of the model. For this purpose, the model of the RAM memory from the H-Inception project [8] was modified to include some assembler instructions in a binary format, as it is shown in Listing 3.10. In the following paragraphs, these assembler instructions are described along with captions of the produced simulation output from the console of the host machine shown in Figure 3.4.

According to the mapping table (Listing 3.5), the address of the GPIO2VCI component was defined as `0x80000000`. In line 3, register 1 will point to address `0x18` of the memory, which has already stored the value `0x80000000` as it can be seen in the last line.

```
1  int mem[1024];
2  ...
3  mem[0x00 >> 2]=0x20010018;  // addi $1,$0, 0x18
4  mem[0x04 >> 2]=0x8c220000;  // lw $2,0($1) : mem[0x18]
5  mem[0x08 >> 2]=0xAC410004;  // sw $1,4($2) : mem[0x80000004] <- 0x18
6  mem[0x0C >> 2]=0x8c430000;  // lw $3,0($2) : mem[0x80000000]
7  mem[0x10 >> 2]=0xAC410004;  // sw $1,4($2) : mem[0x80000004] <- 0x18  //DUMMY
8  mem[0x14 >> 2]=0x8c430000;  // lw $3,0($2) : mem[0x80000000]
9  mem[0x18 >> 2]=0x80000000;
```

Listing 3.10: Assembler instructions hard-coded in RAM memory of the SystemC model.

In line 4, the value on address `0x18` is loaded to register 2, in such way that register 2 will be a pointer to address `0x80000000`. In the simulation output depicted in Figure 3.4a, a "Load" from address `0x18` is requested at time 17.5 ns, and the value `0x80000000` is loaded at time 23 ns.

```
sink_1 @ 17 ns: 0
@17500 ps: LOAD from Address 0x00000018
sink_1 @ 18 ns: 0
sink_1 @ 19 ns: 0
sink_1 @ 20 ns: 0
sink_1 @ 21 ns: 0
sink_1 @ 22 ns: 0
sink_1 @ 23 ns: 0
@23 ns: Loaded value = -2147483648 = 0x80000000
sink_1 @ 24 ns: 0
sink_1 @ 25 ns: 0
sink_1 @ 26 ns: 0
```

(a) "Load" from line 4

```
sink_1 @ 32 ns: 0
@32500 ps: STORE in Address 0x80000004
@32500 ps: STORE value = 24 = 0x00000018
sink_1 @ 33 ns: 0
sink_1 @ 34 ns: 21.0314
sink_1 @ 35 ns: 21.3842
sink_1 @ 36 ns: 21.7158
sink_1 @ 37 ns: 22.0261
sink_1 @ 38 ns: 22.3146
sink_1 @ 39 ns: 22.5811
sink_1 @ 40 ns: 22.8254
sink_1 @ 41 ns: 23.047
```

(b) "Store" from line 5

```
sink_1 @ 44 ns: 23.5749
@44500 ps: LOAD from Address 0x80000000
sink_1 @ 45 ns: 23.7045
sink_1 @ 46 ns: 23.8108
sink_1 @ 47 ns: 23.8935
@47 ns: Loaded value = 23 = 0x00000017
sink_1 @ 48 ns: 23.9526
sink_1 @ 49 ns: 23.9882
sink_1 @ 50 ns: 24
sink_1 @ 51 ns: 23.9882
```

(c) "Load" from line 6

```
sink_1 @ 56 ns: 23.5749
@56500 ps: STORE in Address 0x80000004
@56500 ps: STORE value = 24 = 0x00000018
sink_1 @ 57 ns: 23.422
sink_1 @ 58 ns: 23.246
sink_1 @ 59 ns: 23.047
sink_1 @ 60 ns: 22.8254
sink_1 @ 61 ns: 22.5811
sink_1 @ 62 ns: 22.3146
sink_1 @ 63 ns: 22.0261
sink_1 @ 64 ns: 21.7158
sink_1 @ 65 ns: 21.3842
```

(d) "Store" from line 7

```
sink_1 @ 68 ns: 20.2639
@68500 ps: LOAD from Address 0x80000000
sink_1 @ 69 ns: 19.8499
sink_1 @ 70 ns: 19.4164
sink_1 @ 71 ns: 18.9637
@71 ns: Loaded value = 19 = 0x00000013
sink_1 @ 72 ns: 18.4923
sink_1 @ 73 ns: 18.0027
sink_1 @ 74 ns: 17.4952
sink_1 @ 75 ns: 16.9706
```

(e) "Load" from line 8

Figure 3.4: Simulation output from the the host machine console of the integration of SystemC-AMS and SystemC SoC components.

In line 5, the value `0x18` is stored into address `0x80000004` meaning that the GPIO2VCI component will receive this value as an input from the VCI Crossbar component, who in turn will send this value to the Sine Source component. In Figure 3.4b, a "Store" instruction of the value `0x18` (24 in decimal) in address `0x80000004` is performed at time 32.5 ns. It is important to note that before this value was sent to the GPIO2VCI component, the Sink module connected to the Sine Source was printing a value of 0. But after this "Store" instruction was performed, at time 34 ns, the Sink module starts

printing the values corresponding to the sine wave generated, taking as amplitude the value given to the GPIO2VCI component (`0x18` or 24 in decimal).

In line 6, the value stored in address `0x80000000` will be loaded into register 3. This means that the GPIO2VCI component will receive a read command from the VCI Crossbar component. This command will make the GPIO2VCI component to read the value produced by the Sine Source at that moment and send it back to the VCI Crossbar component. In Figure 3.4c, a "Load" from address `0x80000000` is requested at time 44.5 ns and the value `0x17` (23 in decimal) is loaded at time 47 ns. This value corresponds to the value of the sine wave that is being generated (converted to Integer), which matches to the output printed by the Sink component.

In line 7, the same store instruction as in line 5 is repeated, which will not produce any effect, since the same value `0x18` is being sent (DUMMY instruction). Figure 3.4d shows this "Store" instruction, which will cause no effect to the Sine Source.

But finally in line 8, another read from the GPIO2VCI component is performed via the load instruction from address `0x80000000`. In Figure 3.4e, the "Load" instruction from address `0x80000000` is requested at time 68.5 ns and the value `0x13` (19 in decimal) is loaded at time 71 ns. This value corresponds to the value of the sine wave generated by the Sine Source component and printed by the Sink component.

3.3. Integration of SystemC-AMS modules with a full SoCLib model

In the previous section, a platform integrating SystemC-AMS components and a pedagogic SoCLib SoC was developed. The next step is to integrate SystemC-AMS modules with a full SoCLib model without the use of an OS. For this purpose, an existing SoCLib model from the H-Inception project [8] was used. This model consists of a MIPS32 processor ISS, A 128 KB RAM memory, a TTY console, and a VCI Virtual Micro-network (Vgmn) component. This model uses the "Loader" utility from SoCLib to load cross-compiled C code (Embedded Software) into the RAM memory component.

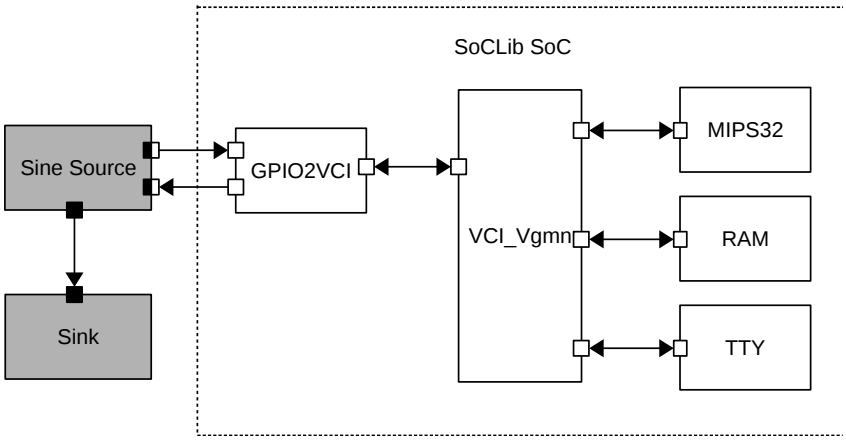


Figure 3.5: Model integrating SystemC-AMS and SoCLib components.

The model was modified in order to include the Sine Source and Sink System-C AMS modules, which are connected via the GPIO2VCI component to the SoCLib SoC model as

it is shown Figure 3.5. The model has a similar behavior as in the previous section, with the difference that now, the SoCLib SoC module will print also messages into the TTY console component based on the embedded software that is programmed to the SoC. Note that this figure shows a simplified version of the model, which does not include all the ports of the GPIO2VCI and the other SoCLib components, but only the most important ports that are used to connect the analog and digital hardware components.

Similar to the model used in the previous section, this SoCLib model uses a mapping table to assign segments of memory to the Hardware components. The declaration and construction of this mapping table is shown in Listing 3.11. The first four segments that are added to the mapping table correspond to the RAM memory component, including the `reset` and exception (`excep`) address spaces, as well as the `data` and instructions (`text`) memory address spaces. The next segment `tty` corresponds to the TTY console component. The last segment `gpio2vci` corresponds to the GPIO2VCI component. In each segment, besides the base address and size parameters, it is important to note the fourth field, which indicates to which target each segment is mapped to. The first four segments (RAM memory segments) are mapped to target 0. The fifth segment `tty` is mapped to target 1 and the sixth segment `gpio2vci` is mapped to target 2. Finally, note that the base address assigned to the `gpio2vci` component is `0xC1200000`. This address was chosen specifically for this model, being an address that does not clash with the addresses of the other components.

```
#define TEXT_BASE       0x00400000
#define TEXT_SIZE       0x00050000
#define RESET_BASE      0xBFC00000
#define RESET_SIZE      0x00010000
#define EXCEP_BASE      0x80000000
#define EXCEP_SIZE      0x00010000
#define DATA_BASE       0x10000000
#define DATA_SIZE       0x00020000
#define TTY_BASE        0xC0200000
#define TTY_SIZE        0x00000040
#define GPIO2VCI_BASE   0xC1200000
#define GPIO2VCI_SIZE   0x00000010
...
  soclib::common::MappingTable maptab(32, IntTab(8), IntTab(8), 0x00300000);
  maptab.add(Segment("reset"   , RESET_BASE   , RESET_SIZE   , IntTab(0), true));
  maptab.add(Segment("excep"   , EXCEP_BASE   , EXCEP_SIZE   , IntTab(0), true));
  maptab.add(Segment("text"    , TEXT_BASE    , TEXT_SIZE    , IntTab(0), true));
  maptab.add(Segment("data"    , DATA_BASE    , DATA_SIZE    , IntTab(0), true));
  maptab.add(Segment("tty"     , TTY_BASE     , TTY_SIZE     , IntTab(1), false));
  maptab.add(Segment("gpio2vci", GPIO2VCI_BASE, GPIO2VCI_SIZE, IntTab(2), false));
```

Listing 3.11: Mapping table for the SoCLib SoC model.

In Listing 3.12, the instantiation of all the components is shown. The RAM memory component (`vcimultiram0`) receives as parameter the SoCLib `loader` utility. There are two important parameters in the VCI Vgmn component (`vgmn`) , the third and foruth parameters, which indicate the number of initiators (1) and targets (3) connected to the VCI component. Also, note that each component receives as parameter the index of the target number to which they are mapped to—e.g. `IntTab(2)` for the `gpio2vci` component.

The Net-List is constructed in a very similar way as in the previous section. In Listing 3.13, only the Net-List for the `gpio2vci` component and the `vgmn` component is shown. As before, the `gpio2vci` component is connected to the SystemC-AMS modules

```
soclib::caba::VciXcacheWrapper<vci_param, soclib::common::Mips32ElIss > cache0("cache0", 0,
maptab,IntTab(0), 4,1,8, 4,1,8);
soclib::common::Loader loader("soft/bin.soft");
soclib::caba::VciRam<vci_param> vcimultiram0("vcimultiram0", IntTab(0), maptab, loader);
soclib::caba::VciMultiTty<vci_param> vcitty("vcitty", IntTab(1), maptab, "vcitty0", NULL);
soclib::caba::VciVgmn<vci_param> vgmn("vgmn",maptab, 1, 3, 3, 8);
soclib::caba::GPIO2VCI<vci_param> gpio2vci0("gpio2vci0", IntTab(2), maptab);
SINE_SOURCE sine0 ("sine0");
SINK sink0("sink0");
```

Listing 3.12: Instantiation of the SoCLib SoC model components.

via the `p_rdata_ams` and `p_wdata_ams` ports, and it is connected to the `vgmn` component via the `p_vci` port. The `vgmn` component connects to the initiator (which in this model is the MIPS32 processor ISS) via the `p_to_initiator[0]` port; and it connects to all the targets via the `p_to_target[n]` port. As the `gpio2vci` component was mapped to target 2, it is connected to the port `p_to_target[2]` of the `vgmn` component.

```
gpio2vci0.p_clk(signal_clk);
gpio2vci0.p_resetn(signal_resetn);
gpio2vci0.p_vci(signal_vci_gpio2vci);
gpio2vci0.p_rdata_ams(s_from_ams);
gpio2vci0.p_wdata_ams(s_to_ams);

vgmn.p_clk(signal_clk);
vgmn.p_resetn(signal_resetn);
vgmn.p_to_initiator[0](signal_vci_m0);
vgmn.p_to_target[0](signal_vci_vcimultiram0);
vgmn.p_to_target[1](signal_vci_tty);
vgmn.p_to_target[2](signal_vci_gpio2vci);
```

Listing 3.13: Net-List of the SoCLib SoC model components.

In order to be able to include the new GPIO2VCI component into SoCLib, a Metadata `.sd` file for this component should be created. Metadata are needed for building SoCLib modules automatically, describe the modules to automated netlisters, and provide extra information about the module [23]. The `.sd` files contains information about the Module that is created, such as the definition and implementation files, class name, ports, etc. Listing A.3 from the Appendix shows the Metadata file `gpio2vci.sd` that was created for the GPIO2VCI component. Along with this Metadata file, the Platform Description file contains the topcell source file name of the model and a list of all the used modules and their parameters. In order to be able to use the GPIO2VCI component with the SoCLib SoC, the `platform_desc` file needed to be modified, as it is shown in Listing A.4 from the Appendix, where line 9 for the `gpio2vci` component was added.

The SoCLib platform was originally designed to work only with SystemC modules. But in this stage of the project, the SystemC-AMS extensions have to be compiled when they are added to the topcell of the model. For this purpose, the SoCLib configuration files need to be modified as well. As it is pointed in the SoCLib documentation [23], there are three ways in which the configuration files can be modified: at the installation directory level, at the user home directory level or at the current directory level. At this stage, modifications at the current directory level were chosen to make sure that the changes would only affect the current model. For this, a `soclib.conf` file was created in the current directory of the model and the necessary SystemC and SystemC-AMS flags

and libraries were configured. The `soclib.conf` file is shown in Listing A.5 from the Appendix.

The SoCLib SoC model that is being used here, already has the necessary libraries to implement C code that can be cross-compiled for the MIPS32 architecture and used as embedded Software for the model, by using the "Loader" utility from SoCLib to load the program into the RAM memory. So finally, to test this second integration approach, a C program was created that could communicate with the GPIO2VCI component and the SystemC-AMS modules in a similar way as it was done in the previous section. Listing 3.14 shows the `main` function of this C code. In Figure 3.6 (a - d), the simulation output from the host machine console is shown. In Figure 3.6e, the output from the TTY terminal is presented. Below, the output of this simulation and the C code program are described to show how the integration between the SystemC-AMS modules and the SoCLib SoC works.

```
1  int main(void)
2  {
3    int * wr_ptr;
4    int * rd_ptr;
5    wr_ptr = (int*)0xC1200000;        //Address of the GPIO2VCI component.
6    rd_ptr = (int*) 0xC1200004;
7    *wr_ptr = 25;
8    printf("Setting amplitude of sine generator to %d\n", *wr_ptr);
9    printf("Reading value of sine generator: %d\n", *rd_ptr);
10   printf("Reading value of sine generator: %d\n", *rd_ptr);
11   printf("Reading value of sine generator: %d\n", *rd_ptr);
12   while (1);
13   return 0;
14 }
```

Listing 3.14: C code of the `main` function from the SoCLib SoC Software.

In lines 3 to 6, two pointers are defined and initialized to the values `0xC1200000` and `0xC1200004` which correspond to the address segments where the GPIO2VCI component is mapped to.

In line 7, the value 25 is written to the address `0xC1200000` pointed by the `wr_ptr` pointer. This will tell the VCI `vgmn` component to send a write command to the GPIO2VCI component, which in turn will send the value through its `p_wdata_ams` port to the Sine Source component. This can bee seen in the host machine console simulation output in Figure 3.6a, where the value 25 is written to address `0xC1200000` at time 1022 ns. Note that before this value is sent to the GPIO2VCI component, the output of the Sink component which is connected to the Sine Source module has a value of 0. But after the GPIO2VCI component writes this value to the Sine Source component, the Sink output starts printing the values of the sine wave generated, using as amplitude the value sent to the GPIO2VCI component. In a similar way, Figure 3.6e shows the output of the SoC TTY console. These messages are printed from the C code program that was loaded to the RAM memory of the SoCLib SoC model. In line 3 of the TTY console, the value written to the GPIO2VCI component is shown as well.

In line 9 of Listing 3.14, the value of address `0xC1200004` pointed by the `rd_ptr` pointer is accessed. This will tell the VCI component to send a read command to the GPIO2VCI component which will in turn read the value from its `p_rdata_ams` port which is connected to the Sine Source component. This is shown in the host machine console simulation output in Figure 3.6b, where the value -12 is read from the GPIO2VCI component

at time 1917 ns. This value corresponds to the output printed by the Sink component at that time, which are the values of the generated sine wave. Similarly, Figure 3.6e shows in line 4 the same value being printed into the TTY console of the SoC model.

Lines 10 and 11 of Listing 3.14 read again values from address `0xC1200004`. In the same way, the GPIO2VCI component will read from its `p_rdata_ams` port the values, as shown in the host machine console simulation output in Figures 3.4c and 3.6d, which correspond to a value of 9 at time 2812 ns and a value of 24 at time 3644 ns. These are the values of the sine wave being read at those specific times. The same values are being printed in the TTY console of the SoC model, as shown in the last two lines of Figure 3.6e.

```
 sink0 @ 1021 ns: 0
 sink0 @ 1022 ns: 0
@1022 ns: Writing to address 0xC1200000 value = 25
 sink0 @ 1023 ns: 16.5328
 sink0 @ 1024 ns: 17.1137
 sink0 @ 1025 ns: 17.6777
 sink0 @ 1026 ns: 18.2242
 sink0 @ 1027 ns: 18.7528
 sink0 @ 1028 ns: 19.2628
```

(a) Host machine console: Write to the GPIO2VCI component.

```
 sink0 @ 1916 ns: -12.0438
 sink0 @ 1917 ns: -12.726
@1917 ns: Reading from address 0xC1200004 value =-12
 sink0 @ 1918 ns: -13.3957
 sink0 @ 1919 ns: -14.0521
 sink0 @ 1920 ns: -14.6946
 sink0 @ 1921 ns: -15.3227
 sink0 @ 1922 ns: -15.9356
 sink0 @ 1923 ns: -16.5328
```

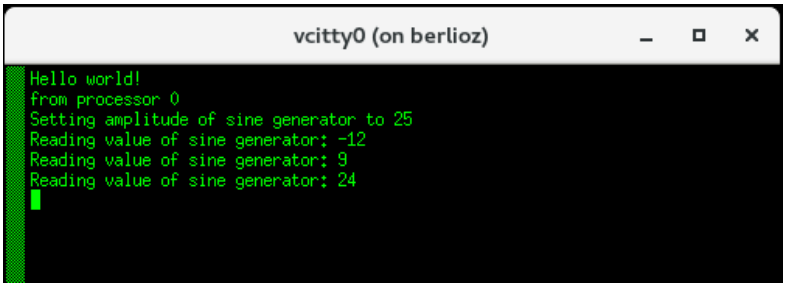
(b) Host machine console: Read from the GPIO2VCI component.

```
 sink0 @ 2811 ns: 8.46845
 sink0 @ 2812 ns: 9.20311
@2812 ns: Reading from address 0xC1200004 value =9
 sink0 @ 2813 ns: 9.9287
 sink0 @ 2814 ns: 10.6445
 sink0 @ 2815 ns: 11.3498
 sink0 @ 2816 ns: 12.0438
 sink0 @ 2817 ns: 12.726
 sink0 @ 2818 ns: 13.3957
```

(c) Host machine console: Read from the GPIO2VCI component.

```
 sink0 @ 3643 ns: 24.3979
 sink0 @ 3644 ns: 24.5572
@3644 ns: Reading from address 0xC1200004 value =24
 sink0 @ 3645 ns: 24.6922
 sink0 @ 3646 ns: 24.8029
 sink0 @ 3647 ns: 24.889
 sink0 @ 3648 ns: 24.9507
 sink0 @ 3649 ns: 24.9877
 sink0 @ 3650 ns: 25
```

(d) Host machine console: Read from the GPIO2VCI component.



(e) TTY console output.

Figure 3.6: Simulation of the integration of SystemC-AMS and SoCLib SoC components.

4. Time synchronization between TDF and DE MoCs

4.1. Detection of time synchronization issues

As mentioned in Chapter 1, when executing a SystemC-AMS simulation of a model, synchronization issues between the TDF and the DE MoCs may arise, specially when dealing with multi-rate TDF blocks that are connected to DE blocks by means of TDF converter ports. According to [24], when a SystemC-AMS simulation is being executed, the execution of the SystemC DE simulation kernel is blocked while the SystemC-AMS simulation kernel is running. As a consequence, during this period the DE simulation time (t_{DE}) does not advance at all, while the TDF simulation time (t_{TDF}) is running according to the timesteps of the TDF modules and ports. When there is an access to a TDF converter port (which connects one DE and one TDF block), the SystemC-AMS simulation kernel is interrupted and yields to the SystemC DE simulation kernel. In this way, the t_{DE} advances until it is equal to the t_{TDF}. In general the t_{TDF} runs ahead of the t_{DE}, such that Equation 4.1 should always hold.

$$t_{\mathrm{TDF}} \geq t_{\mathrm{DE}} \tag{4.1}$$

In some scenarios, the t_{DE} may be greater than the t_{TDF} which may generate causality problems during the simulation of the model.

Here, three synchronization operations are defined.

1. Periodic synchronization operation: It occurs when a period of a TDF cluster has completed. A causality check is done using Equation 4.1. The SystemC-AMS simulation kernel is interrupted and yields to the SystemC DE simulation kernel. In consequence, the t_{DE} advances until it is equal to the t_{TDF}.

2. Read synchronization operation: It occurs when an access to an input converter port occurs. A causality check is done using Equation 4.1. The SystemC-AMS simulation kernel is interrupted and yields to the SystemC DE simulation kernel. In consequence, the t_{DE} advances until it is equal to the t_{TDF}.

3. Write synchronization operation: It occurs when an access to an output converter port occurs. A causality check is done using Equation 4.1. But in this case, the SystemC-AMS simulation kernel is not interrupted, hence the t_{DE} does not advance.

In the following subsections, four different possible scenarios are presented and analyzed in order to show when a time synchronization issue may occur, depending on the sequence of accesses to the TDF converter ports. A simple example adapted from [21, 7] will be used to create the TDF models and execute the simulations that will help to describe the different scenarios.

4.1.1. Access to input converter port before an access to output converter port

For this first scenario, the model shown in Figure 4.1 was used. It consists of two TDF modules **A** and **B** connected to two DE modules **X** and **Y** through the signals **sig1** and **sig3** respectively by means of the TDF converter ports **A.in** and **B.out**. For

each module, the Module-Timestep (**Tm**) and the port parameters Rate (**R**), Delay (**D**) and Port-Timestep (**Tp**) are given. According to the module and port parameters, the determined execution order or static schedule of the TDF modules for one period of this model is **ABABB**, as explained in Appendix C. The SystemC-AMS simulation output of this model is shown in Figure 4.2.

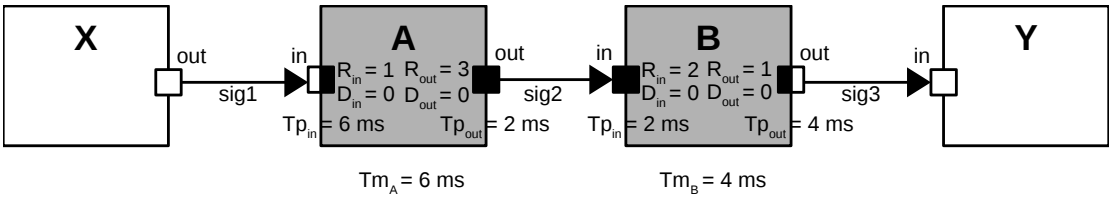
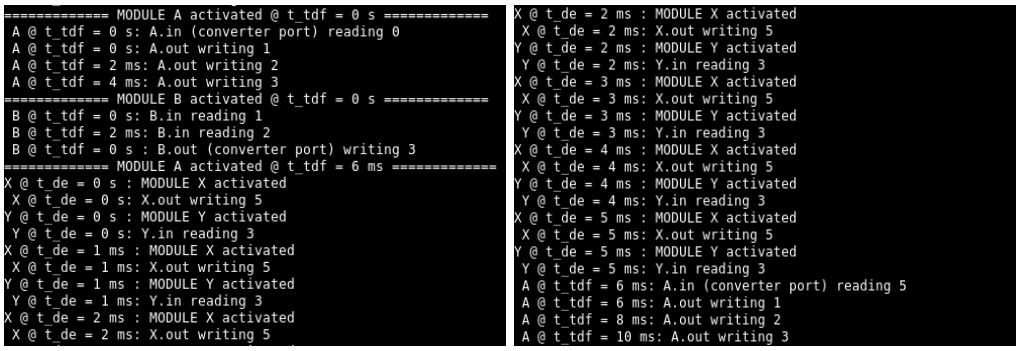


Figure 4.1: TDF-DE model accessing input converter port before accessing output converter port. (Adapted from [7]).



(a) **(b)**

```
A @ 6 ms: A.in (converter port) reading 5
A @ 6 ms : A.out writing 1
A @ 8 ms : A.out writing 2
A @ 10 ms : A.out writing 3
============= MODULE B activated =============
B @ 4 ms: B.in reading 3
B @ 6 ms: B.in reading 1

Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scams/impl/core
/sca_solver_base.cpp line: 529 current sca-time: 4 ms current sc-time: 6 ms sca-next-time:   8 ms
 insert da delay of at least: 2 ms in: B.out
```

(c) Detection of time synchronization issue.

Figure 4.2: SystemC-AMS simulation of model accessing input converter port before accessing output converter port.

Step	Sync. Type	t_{TDF} (ms)	t_{DE} (ms)	$t_{\text{TDF}} \geq t_{\text{DE}}$	New t_{DE} (ms)
1	Read	$t_{\text{TDF}_{A.in}} = 0$	0	True	0
2	–	–	–	–	–
3	–	–	–	–	–
4	Write	$t_{\text{TDF}_{B.out}} = 0$	0	True	0
5	Read	$t_{\text{TDF}_{A.in}} = 6$	0	True	6
6	–	–	–	–	–
7	–	–	–	–	–
8	Write	$t_{\text{TDF}_{B.out}} = 4$	6	**False**	–

Table 4.1: TDF and DE simulation time tracking for model accessing input converter port before accessing output converter port.

Below, the details of the execution of this model's simulation are shown. Table 4.1 shows how the t_{TDF} and t_{DE} advance when a synchronization operation occurs, according to each step of the below description. It also shows if Equation 4.1 holds or if a causality problem is generated.

1. At the start of the simulation, t_{DE}, t_{TDF_A}, and t_{TDF_B} are at 0 ms. Module **A** activates at $t_{\mathrm{TDF}_A} = 0$ ms, but for module **A** to be executed, an access to the input converter port **A.in** will occur, generating a read synchronization operation. In consequence, the SystemC DE simulation kernel is executed and t_{DE} advances to 0 ms. This step is not really shown in Figure 4.2 because when the access to the input converter port **A.in** happens, t_{DE} is already equal to $t_{\mathrm{TDF}_{A.in}}$.

2. After this, module **A** reads the available sample from the input converter port **A.in** at 0 ms, executes its internal functions and writes three samples into the output port **A.out** with timestamps of 0 ms, 2 ms and 4 ms respectively. This can be seen in the upper part of Figure 4.2a.

3. Module **B** is activated at $t_{\mathrm{TDF}_B} = 0$ ms. As shown in Figure 4.2a, it reads two of the three available samples from the input port **B.in** at 0 ms and 2 ms respectively, executes its internal functions and writes one sample into the output converter port **B.out** at 0 ms.

4. Since an access to the output converter port **B.out** occurred at 0 ms, a write synchronization operation is generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{B.out}} = 0$ ms and $t_{\mathrm{DE}} = 0$ ms, but as shown in Figure 4.2a, the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

5. According to the schedule, the next module to be executed is **A**. Since the timestep of the module is $\mathrm{Tm}_A = 6$ ms, it activates at $t_{\mathrm{TDF}_A} = 6$ ms as Figure 4.2a shows. But once again in order to be executed, an access to the input converter port **A.in** will occur at 6 ms and a read synchronization operation will be generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{A.in}} = 6$ ms and $t_{\mathrm{DE}} = 0$ ms. As it can be seen in the lower part of Figure 4.2a and the beginning of Figure 4.2b, the SystemC DE simulation kernel is executed and t_{DE} advances until it reaches $t_{\mathrm{TDF}_{A.in}}$; that is $t_{\mathrm{DE}} = 6$ ms. Figure 4.2b shows that the **X** and **Y** modules execute up to $t_{\mathrm{DE}} = 5$ ms. After this, the t_{DE} advances to 6 ms and then the SystemC-AMS simulation kernel takes control.

6. Now module **A** reads the available sample from the input converter port **A.in** at 6 ms, executes its internal functions and writes three samples into the output port **A.out** with timestamps of 6 ms, 8 ms and 10 ms respectively, as it is shown in the lower part of Figure 4.2b or the upper part of Figure 4.2c.

7. The next module to be activated according to the schedule is module **B**. Its timestep is $\mathrm{Tm}_B = 4$ ms, hence it is activated at $t_{\mathrm{TDF}_B} = 4$ ms. It reads two of the four available samples from the input port **B.in** at 4 ms and 6 ms respectively, as shown in Figure 4.2c. It executes its internal functions and tries to write one sample into the output converter port **B.out** at 4 ms.

8. Since an access to the output converter port **B.out** occurred at 4 ms, a write synchronization operation would be required. In step 5, a read synchronization

operation was performed, so $t_{\mathrm{DE}} = 6$ ms. But since $t_{\mathrm{TDF}_{B.out}} = 4$ ms, Equation 4.1 does not hold and a causality problem is generated. This is depicted in the last lines of Figure 4.2c, where a time synchronization problem was detected by the SystemC-AMS simulator. We can see in this figure that the $t_{\mathrm{TDF}_{B.out}} = 4$ ms (current sca-time: 4 ms) and the $t_{\mathrm{DE}} = 6$ ms (current sc-time: 6 ms).

4.1.2. Access to output converter port before an access to input converter port

In this second scenario, a similar model is used. The difference is that the type of converter ports are modified as shown in Figure 4.3. The TDF module **A** has an output converter port **A.out_de** connected to the DE module **X**, while the TDF module **B** has an input converter port **B.in_de** connected to the DE module **Y**. The other parameters are the same as in the previous scenario. But my making these changes, this scenario will test the behavior of the SystemC-AMS simulator when accessing an output converter port before accessing an input converter port. Similarly, the SystemC-AMS simulation output is shown in Figure 4.4

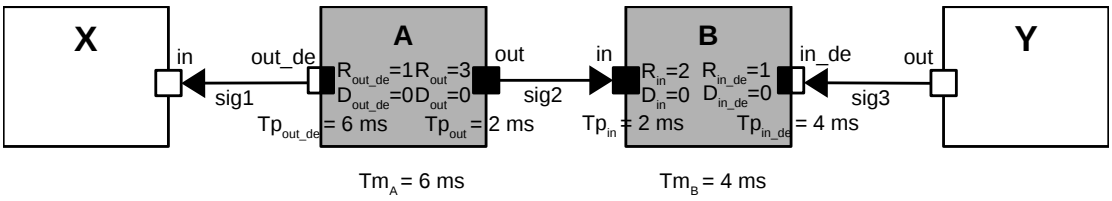
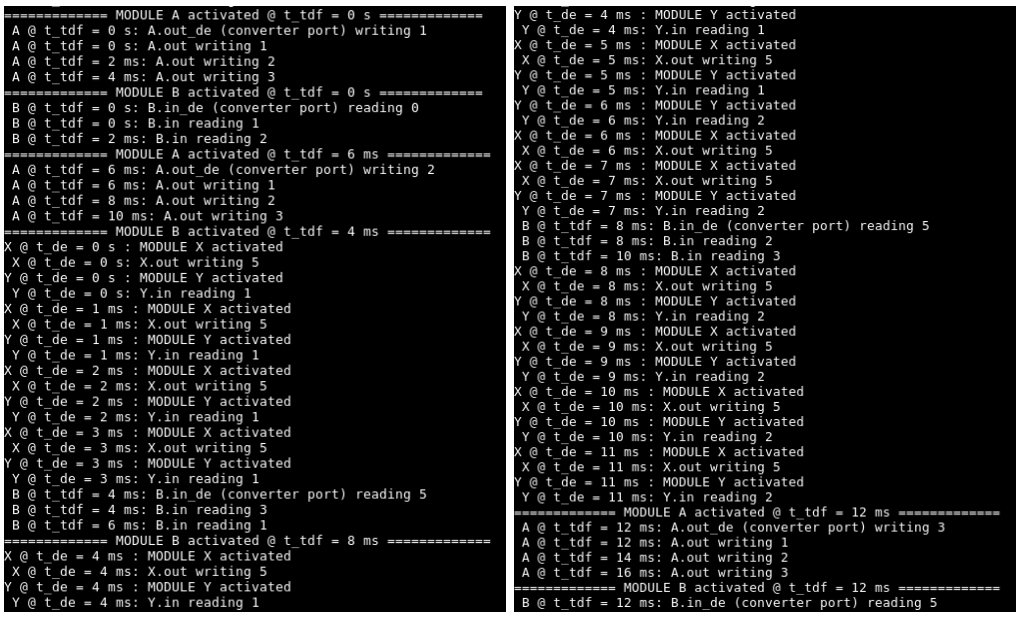


Figure 4.3: TDF-DE model accessing output converter port before accessing input converter port. (Adapted from [7]).



(a) **(b)**

Figure 4.4: SystemC-AMS simulation of model accessing output converter port before accessing input converter port.

The details of the execution of this model are described below. Table 4.2 presents how the t_{TDF} and t_{DE} advance when a synchronization operation occurs, according to each step of the below description. It also shows if Equation 4.1 holds or if a causality problem is generated.

Step	Sync. Type	t_{TDF} (ms)	t_{DE} (ms)	$t_{\mathrm{TDF}} \geq t_{\mathrm{DE}}$	New t_{DE} (ms)
1	Write	$t_{\mathrm{TDF}_{A.out\_de}} = 0$	0	True	0
2	Read	$t_{\mathrm{TDF}_{B.in\_de}} = 0$	0	True	0
3	–	–	–	–	–
4	Write	$t_{\mathrm{TDF}_{A.out\_de}} = 6$	0	True	0
5	Read	$t_{\mathrm{TDF}_{B.in\_de}} = 4$	0	True	4
6	–	–	–	–	–
7	Read	$t_{\mathrm{TDF}_{B.in\_de}} = 8$	4	True	8
8	–	–	–	–	–
9	Periodic	$t_{\mathrm{TDF}} = 12$	8	–	12

Table 4.2: TDF and DE simulation time tracking for model accessing output converter port before accessing input converter port.

1. At the start of the simulation, t_{DE}, t_{TDF_A}, and t_{TDF_B} are at 0 ms. Module **A** activates at $t_{\mathrm{TDF}_A} = 0$ ms. It executes its internal functions, writes one sample into the output converter port **A.out_de** at 0 ms and writes three samples into the output port **A.out** with timestamps of 0 ms, 2 ms and 4 ms respectively, as it is shown in Figure 4.4a. Since an access to the output converter port **A.out_de** occurred at 0 ms, a write synchronization operation is generated. Equation 4.1 holds because $t_{\mathrm{TDF}_{A.out\_de}} = 0$ ms and $t_{\mathrm{DE}} = 0$ ms, but as it can be seen in Figure 4.4a, the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

2. Then, Module **B** is activated at $t_{\mathrm{TDF}_B} = 0$ ms. For module **B** to be executed, an access to the input converter port **B.in_de** will occur, generating a read synchronization operation. In consequence, the SystemC DE simulation kernel is executed and t_{DE} advances to 0 ms. This step is not shown in Figure 4.4 because when the access to the input converter port **B.in_de** happens, t_{DE} is already equal to $t_{\mathrm{TDF}_{B.in\_de}}$.

3. Module **B** executes its internal functions and reads the sample available at the input converter port **B.in_de** at $t_{\mathrm{TDF}_{B.in\_de}} = 0$ ms and two of the three samples available at the input port **B.in** at $t_{\mathrm{TDF}_{B.in}}$ equals to 0 ms and 2 ms respectively.

4. According to the schedule, the next module to be executed is **A**. Since the timestep of the module is $\mathrm{Tm}_A = 6$ ms, it activates at $t_{\mathrm{TDF}_A} = 6$ ms, as Figure 4.4a shows. It executes its internal functions and again, writes one sample into the output converter port **A.out_de** at 6 ms and writes three samples into the output port **A.out** with timestamps of 6 ms, 8 ms and 10 ms respectively. A write synchronization operation is generated because the output converter port **A.out_de** was accessed. Equation 4.1 still holds because $t_{\mathrm{DE}} = 0$ ms. And as Figure 4.4a shows, the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

5. Now, module **B** is activated at $t_{\mathrm{TDF}_B} = 4$ ms, but in order to be executed, an access to the input converter port **B.in_de** will occur at 4 ms and a read synchronization

operation will be generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{B.in\_de}} = 4$ ms and $t_{\mathrm{DE}} = 0$ ms. Figure 4.4a shows that the SystemC DE simulation kernel is executed and t_{DE} advances until it reaches $t_{\mathrm{TDF}_{B.in\_de}}$; that is $t_{\mathrm{DE}} = 4$ ms. Figure 4.4a shows that the **X** and **Y** modules execute up to $t_{\mathrm{DE}} = 3$ ms. After this, the t_{DE} advances to 4 ms and then the SystemC-AMS simulation kernel takes control.

6. Module **B** executes its internal functions and reads the available sample from the input converter port **B.in_de** at $t_{\mathrm{TDF}_{B.in\_de}} = 4$ ms. It reads two of the four samples available at the input port **B.in** at $t_{\mathrm{TDF}_{B.in}}$ equals to 4 ms and 6 ms respectively, as Figure 4.4a shows.

7. The next module to be activated according to the schedule is module **B**. It activates at $t_{\mathrm{TDF}_B} = 8$ ms, as it can be seen in the bottom part of Figure 4.4a. In order to be executed, an access to the input converter port **B.in_de** will occur at 8 ms and a read synchronization operation will be generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{B.in\_de}} = 8$ ms and $t_{\mathrm{DE}} = 4$ ms. Now, as it is shown in Figures 4.4a and 4.4b, the SystemC DE simulation kernel is executed and t_{DE} advances until it reaches $t_{\mathrm{TDF}_{B.in\_de}} = 8$ ms. Again, the **X** and **Y** modules execute up to $t_{\mathrm{DE}} = 7$ ms. After this, the t_{DE} advances to 8 ms and then the SystemC-AMS simulation kernel takes control.

8. Finally, module **B** executes its internal functions, reads the available sample from the input converter port **B.in_de** at $t_{\mathrm{TDF}_{B.in\_de}} = 8$ ms and reads the last two available samples at the input port **B.in** at $t_{\mathrm{TDF}_{B.in}}$ equals to 8 ms and 10 ms respectively.

9. At this point, one period of the schedule of the model has been executed. Hence, a periodic synchronization operation occurs. The SystemC DE simulation kernel executes and t_{DE} advances until it reaches the new t_{TDF} for the start of a new TDF cluster period; that is until $t_{\mathrm{DE}} = 12$ ms. This is shown the bottom of Figure 4.4b, where a new period starts.

It is important to notice that in this scenario there was no time synchronization issues even if same TDF module parameters as in the previous scenario were used. The only thing that was modified, were the type of converter ports used in each of the TDF modules. So this behavior was present due to the fact that the t_{DE} only advanced after a read synchronization operation occurred or when a period of the TDF cluster was completed.

4.1.3. Access to output converter port before an access to another output converter port

In the following scenario, the case when two output converter ports are accessed one after another will be analyzed. The same model used in the previous scenarios will be modified in order that the converter port of each of the two TDF modules will be an output converter port. All the other module parameters are being kept the same. This is depicted in Figure 4.5, where the module **A** has an output converter port **A.out_de** and the module **B** has another output converter port **B.out_de**. Once again, the SystemC-AMS simulation output of this model is presented in Figure 4.6.

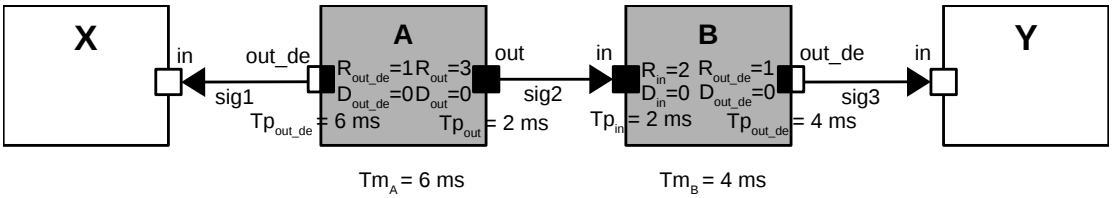
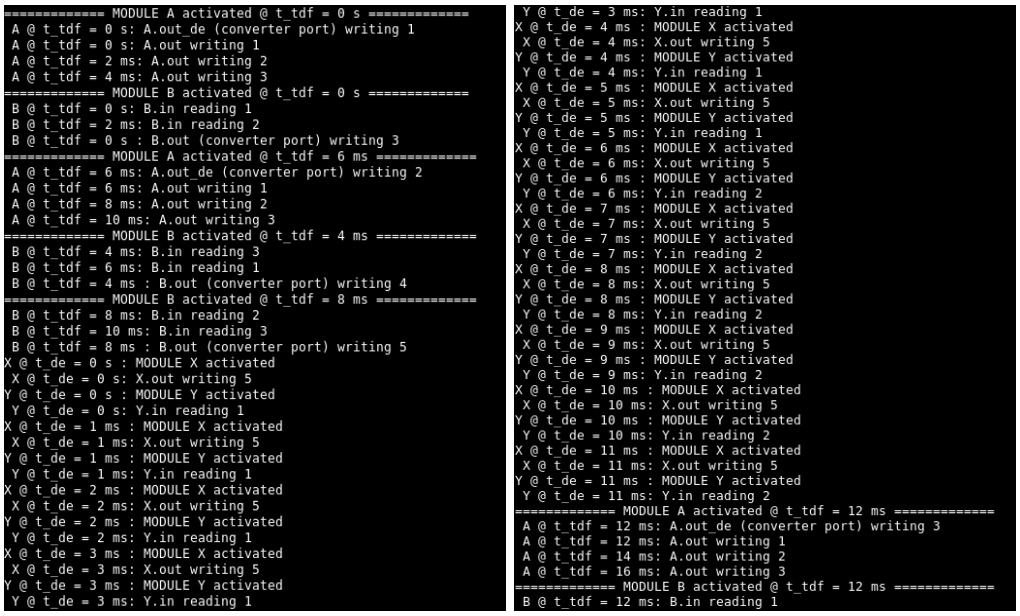


Figure 4.5: TDF-DE model accessing output converter port before accessing another output converter port. (Adapted from [7]).



(a) **(b)**

Figure 4.6: SystemC-AMS simulation of model accessing output converter port before accessing another output converter port.

Step	Sync. Type	t_{TDF} (ms)	t_{DE} (ms)	$t_{\text{TDF}} \geq t_{\text{DE}}$	New t_{DE} (ms)
1	–	–	–	–	–
2	Write	$t_{\text{TDF}_{A.out\_de}} = 0$	0	True	0
3	–	–	–	–	–
4	Write	$t_{\text{TDF}_{B.out\_de}} = 0$	0	True	0
5	–	–	–	–	–
6	Write	$t_{\text{TDF}_{A.out\_de}} = 6$	0	True	0
7	–	–	–	–	–
8	Write	$t_{\text{TDF}_{B.out\_de}} = 4$	0	True	0
9	–	–	–	–	–
10	Write	$t_{\text{TDF}_{B.out\_de}} = 8$	0	True	0
11	Periodic	$t_{\text{TDF}} = 12$	0	–	12

Table 4.3: TDF and DE simulation time tracking for model accessing output converter port before accessing another output converter port.

The description of the execution of the SystemC simulation output is presented below. As in the previous scenarios, Table 4.3 shows how the t_{TDF} and t_{DE} advance when a

synchronization operation occurs, according to each step of the below description. It also shows if Equation 4.1 holds or if a causality problem is generated.

1. At the beginning of the simulation, t_{DE}, t_{TDF_A}, and t_{TDF_B} are at 0 ms. Module **A** activates at $t_{\mathrm{TDF}_A} = 0$ ms. It executes its internal functions, writes one sample into the output converter port **A.out_de** at 0 ms and writes three samples into the output port **A.out** with timestamps of 0 ms, 2 ms and 4 ms respectively. This is shown in the upper part of Figure 4.6a.

2. Since an access to the output converter port **A.out_de** occurred at 0 ms, a write synchronization operation is generated. Equation 4.1 holds because $t_{\mathrm{TDF}_{A.out\_de}} = 0$ ms and $t_{\mathrm{DE}} = 0$ ms, but as it can be seen in Figure 4.6a, the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

3. Module **B** is activated at $t_{\mathrm{TDF}_B} = 0$ ms. As shown in Figure 4.6a, it reads two of the three available samples from the input port **B.in** at 0 ms and 2 ms respectively, executes its internal functions and writes one sample into the output converter port **B.out_de** at 0 ms.

4. Because an access to the output converter port **B.out_de** occurred at 0 ms, a write synchronization operation is generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{B.out\_de}} = 0$ ms and $t_{\mathrm{DE}} = 0$ ms. But again, the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

5. According to the schedule, the next module to be executed is **A**. Since the timestep of the module is $\mathrm{Tm}_A = 6$ ms, it activates at $t_{\mathrm{TDF}_A} = 6$ ms, as Figure 4.6a shows. It executes its internal functions and again, writes one sample into the output converter port **A.out_de** at 6 ms and writes three samples into the output port **A.out** with timestamps of 6 ms, 8 ms and 10 ms respectively.

6. A write synchronization operation is generated because the output converter port **A.out_de** was accessed at 6 ms. Equation 4.1 still holds because $t_{\mathrm{DE}} = 0$ ms. And as Figure 4.6a shows, the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

7. The next module to be activated according to the schedule is module **B**. Its timestep is $\mathrm{Tm}_B = 4$ ms, hence it is activated at $t_{\mathrm{TDF}_B} = 4$ ms. It reads two of the four available samples from the input port **B.in** at 4 ms and 6 ms respectively, as shown in Figure 4.6a. It executes its internal functions and tries to write one sample into the output converter port **B.out_de** at 4 ms.

8. This access to the output converter port **B.out_de** at 4 ms generates a write synchronization operation. Equation 4.1 still holds because t_{DE} has not advanced yet. And as it is shown in Figure 4.6a, the SystemC DE simulation kernel does not execute and t_{DE} still does not advance.

9. It is turn again for module **B** to be executed according to the schedule. It is activated at $t_{\mathrm{TDF}_B} = 8$ ms. It reads the last two available samples at the input port **B.in** at $t_{\mathrm{TDF}_{B.in}}$ equals to 8 ms and 10 ms respectively. And as shown in Figure 4.6a, it writes one sample into the output converter port **B.out_de** at 8 ms.

10. Since an access to the output converter port **B.out_de** occurred at 8 ms, a write synchronization operation is generated. Equation 4.1 still holds, but the SystemC DE simulation kernel does not execute and t_{DE} still does not advance.

11. At this point, one period of the TDF cluster has been executed. A periodic synchronization operation is generated, and as it can be seen in Figures 4.6a and 4.6b, the SystemC DE simulation kernel finally executes and t_{DE} advances from 0 ms until it reaches the new t_{TDF} for the start of a new TDF cluster period; that is until $t_{\mathrm{DE}} = 12$ ms. At the bottom of Figure 4.6b, the execution of a new TDF cluster period at $t_{\mathrm{TDF}} = 12$ ms occurs. And at this point in time, the SystemC DE simulation kernel is interrupted.

During the analysis of this scenario, no synchronization issues occurred, because the t_{DE} didn't advance when write synchronization operations took place. The SystemC DE kernel only executed and the t_{DE} advanced only when a period of the TDF cluster was completed.

4.1.4. Access to input converter port before an access to another input converter port

In this last scenario, the case when two input converter ports are accessed one after another is analyzed. A similar model will be used for this purpose, with the difference that the converter ports of the TDF modules **A** and **B** will be input converter ports in both cases. Figure 4.7 shows this model, where module **A** has an input converter port **A.in** and module **B** has another input converter port **B.in_de**. The SystemC-AMS simulation output of this model is shown in Figure 4.8.

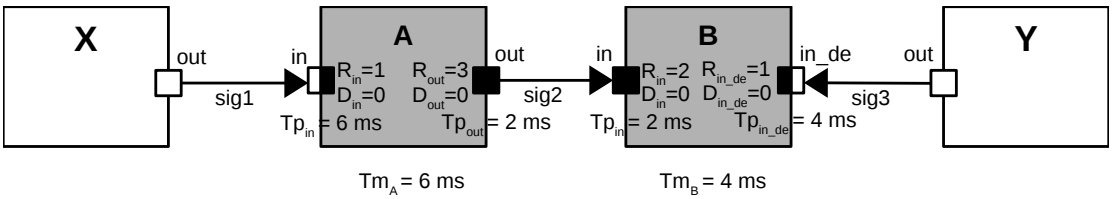
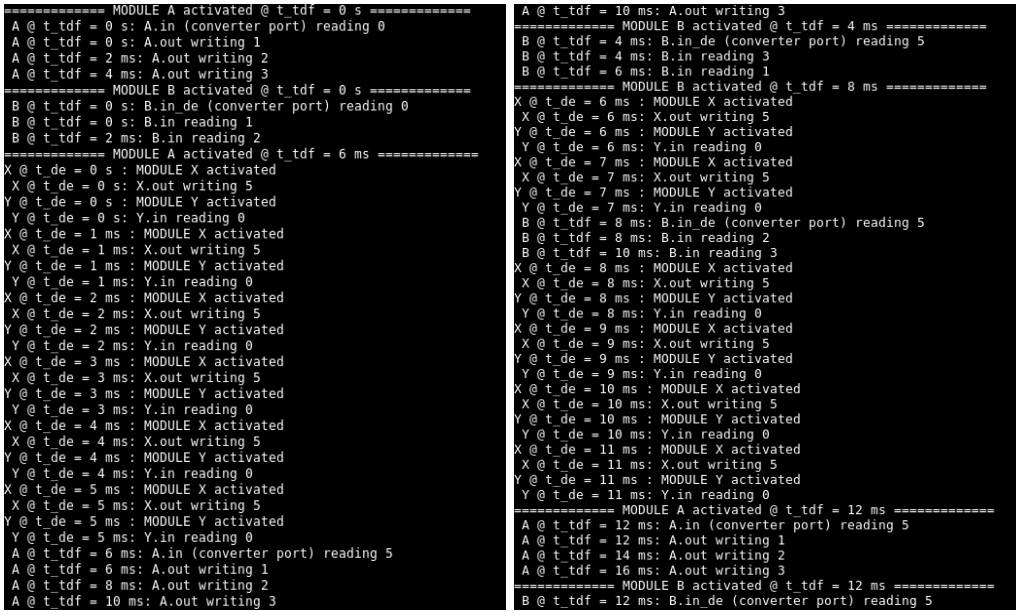


Figure 4.7: TDF-DE model accessing input converter port before accessing another input converter port. (Adapted from [7]).

Below, an explanation of the execution of the SystemC-AMS simulation of this model is presented. Table 4.4 shows how the t_{TDF} and t_{DE} advance depending on the synchronization operations that may occur. It also shows if Equation 4.1 holds, denoting if a synchronization problem has occurred or not.

1. At the start of the simulation, t_{DE}, t_{TDF_A}, and t_{TDF_B} are at 0 ms. Module **A** activates at $t_{\mathrm{TDF}_A} = 0$ ms, but for module **A** to be executed, an access to the input converter port **A.in** will occur, generating a read synchronization operation. In consequence, the SystemC DE simulation kernel is executed and t_{DE} advances to 0 ms. This step is not shown in Figure 4.8 because when the access to the input converter port **A.in** happens, t_{DE} is already equal to $t_{\mathrm{TDF}_{A.in}}$.



(a) **(b)**

Figure 4.8: SystemC-AMS simulation of model accessing input converter port before accessing another input converter port.

2. After this, module **A** reads the available sample from the input converter port **A.in** at 0 ms, executes its internal functions and writes three samples into the output port **A.out** with timestamps of 0 ms, 2 ms and 4 ms respectively. This is shown in the upper part of Figure 4.8a.

3. Then, Module **B** is activated at $t_{\mathrm{TDF}_B} = 0$ ms. For module **B** to be executed, an access to the input converter port **B.in_de** will occur, generating a read synchronization operation. In consequence, the SystemC DE simulation kernel is executed and t_{DE} advances to 0 ms. This step is not shown in Figure 4.4 because when the access to the input converter port **B.in_de** happens, t_{DE} is already equal to $t_{\mathrm{TDF}_{B.in\_de}}$.

4. Module **B** executes its internal functions and reads the sample available at the input converter port **B.in_de** at $t_{\mathrm{TDF}_{B.in\_de}} = 0$ ms and two of the three samples available at the input port **B.in** at $t_{\mathrm{TDF}_{B.in}}$ equals to 0 ms and 2 ms respectively.

5. According to the schedule, the next module to be executed is **A**. Since the timestep of the module is $\mathrm{Tm}_A = 6$ ms, it activates at $t_{\mathrm{TDF}_A} = 6$ ms as Figure 4.8a shows. But once again in order to be executed, an access to the input converter port **A.in** will occur at 6 ms and a read synchronization operation will be generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{A.in}} = 6$ ms and $t_{\mathrm{DE}} = 0$ ms. As Figure 4.8a shows, the SystemC DE simulation kernel is executed and t_{DE} advances until it reaches $t_{\mathrm{TDF}_{A.in}}$; that is $t_{\mathrm{DE}} = 6$ ms. Figure 4.8a shows that the **X** and **Y** modules execute up to $t_{\mathrm{DE}} = 5$ ms. After this, the t_{DE} advances to 6 ms and then the SystemC-AMS simulation kernel takes control.

6. Now module **A** reads the available sample from the input converter port **A.in** at 6 ms, executes its internal functions and writes three samples into the output port

Step	Sync. Type	t_{TDF} (ms)	t_{DE} (ms)	$t_{\text{TDF}} \geq t_{\text{DE}}$	New t_{DE} (ms)
1	Read	$t_{\text{TDF}_{A.in}} = 0$	0	True	0
2	–	–	–	–	–
3	Read	$t_{\text{TDF}_{B.in\_de}} = 0$	0	True	0
4	–	–	–	–	–
5	Read	$t_{\text{TDF}_{A.in}} = 6$	0	True	6
6	–	–	–	–	–
7	Write	$t_{\text{TDF}_{B.in\_de}} = 4$	6	**False-no sync. issue**	6
8	–	–	–	–	–
9	Write	$t_{\text{TDF}_{B.in\_de}} = 8$	6	True	8
10	–	–	–	–	–
11	Periodic	$t_{\text{TDF}} = 12$	8	–	12

Table 4.4: TDF and DE simulation time tracking for model accessing input converter port before accessing another input converter port.

A.out with timestamps of 6 ms, 8 ms and 10 ms respectively, as it is shown in the lower part of Figure 4.8a.

7. The next module to be activated according to the schedule is module **B**. Its timestep is $\text{Tm}_B = 4$ ms, hence it is activated at $t_{\text{TDF}_B} = 4$ ms. But in order to be executed, an access to the input converter port **B.in_de** will occur at 4 ms and a read synchronization operation will be generated. Notice here that Equation 4.1 will NOT hold, since $t_{\text{TDF}_{B.in\_de}} = 4$ ms and $t_{\text{DE}} = 6$ ms. But as it can be seen from the simulation output in Figure 4.8b, NO synchronization issue was generated. In this case the t_{DE} should advance to 4 ms, but since it is already equal to 6 ms, no action is taken.

8. So module **B** executes its internal functions and reads the available sample from the input converter port **B.in_de** at $t_{\text{TDF}_{B.in\_de}} = 4$ ms. It reads two of the four samples available at the input port **B.in** at $t_{\text{TDF}_{B.in}}$ equals to 4 ms and 6 ms respectively, as Figure 4.8b shows.

9. Then, it is turn for module **B** to be activated according to the schedule. It activates at $t_{\text{TDF}_B} = 8$ ms, as Figure 4.8b shows. In order to be executed, an access to the input converter port **B.in_de** will occur at 8 ms and a read synchronization operation will be generated. In this case Equation 4.1 holds since $t_{\text{TDF}_{B.in\_de}} = 8$ ms and $t_{\text{DE}} = 6$ ms. Now, as it is shown in Figure 4.8b, the SystemC DE simulation kernel is executed and t_{DE} advances until it reaches $t_{\text{TDF}_{B.in\_de}} = 8$ ms. Again, the **X** and **Y** modules execute up to $t_{\text{DE}} = 7$ ms. After this, the t_{DE} advances to 8 ms and then the SystemC-AMS simulation kernel takes control.

10. Finally, module **B** executes its internal functions, reads the available sample from the input converter port **B.in_de** at $t_{\text{TDF}_{B.in\_de}} = 8$ ms and reads the last two available samples at the input port **B.in** at $t_{\text{TDF}_{B.in}}$ equals to 8 ms and 10 ms respectively.

11. At this point, one period of the TDF cluster has been executed. A periodic synchronization operation is generated, and as Figure 4.8b shows, the SystemC DE simulation kernel finally executes and t_{DE} advances from 8 ms until it reaches the

new t_{TDF} for the start of a new TDF cluster period; that is until $t_{\mathrm{DE}} = 12$ ms. At the bottom of Figure 4.8b, the execution of a new TDF cluster period at $t_{\mathrm{TDF}} = 12$ ms occurs. And at this point in time, the SystemC DE simulation kernel is interrupted.

From the analysis of this scenario, even if Equation 4.1 failed to hold during step 7, the SystemC-AMS simulator didn't generate any synchronization issues. This means that the time synchronization check happens, but if Equation 4.1 does not hold, the SystemC DE simulation kernel will not execute but no synchronization issue will be generated. If it holds, the SystemC DE simulation kernel will take control and execute normally, and the t_{DE} will advance.

4.1.5. Preliminary conclusions

Considering the previous four scenarios' analysis, two things can be concluded:

1. Only in two cases the SystemC-AMS simulation kernel is interrupted to yield to the SystemC DE simulation kernel and in consequence the t_{DE} advances:

   a) When there is an access to a TDF input converter port and Equation 4.1 holds.

   b) When one period of the TDF cluster has finished executing.

2. Time synchronization issues between DE and TDF MoCs only occur when there was an access to an input converter port that advanced the t_{DE} further than the t_{TDF} of the actual output converter port that is being accessed. The reason is that the t_{TDF} of the output converter ports needs to be always greater or equal than the t_{DE}, according to Equation 4.1. In other words, time synchronization issues may only occur when there is a read synchronization operation before a write synchronization operation. The analysis shows that there are no causality problems in any other scenario—i.e. read before read, write before write, or write before read synchronization operations.

4.2. Avoidance of time synchronization issues

From the previous analysis of the first scenario in Subsection 4.1.1, in order to avoid the time synchronization issue that was detected, the sample that is written into the output converter port **B.out** should be written at least at $t_{\mathrm{TDF}_{B.out}} = 6$ ms, such that $t_{\mathrm{TDF}_{B.out}} \geq t_{\mathrm{DE}}$ holds. We see that the timestamp when the sample is written into **B.out** should be shifted by

$$t_{\mathrm{DE}} - t_{\mathrm{TDF}_{B.out}} = 6 \text{ ms - } 4 \text{ ms} = 2 \text{ ms}.$$

This is similarly shown in the SystemC-AMS simulation output from Figure 4.2c: "insert delay of at least: 2 ms". To achieve this, we can make use of the Delay parameter of a

TDF converter port's module. In general, the minimum required Delay value in a port **p** from a module **M** ($D_{req_{M.p}}$) can be determined by Equation 4.2.

$$D_{req_{M.p}} = \left\lceil \frac{t_{\mathrm{DE}} - t_{\mathrm{TDF}_{M.p}}}{\mathrm{Tp}_{M.p}} \right\rceil \tag{4.2}$$

To solve the synchronization issue using Equation 4.2, the minimum required delay in port **B.out** should be

$$D_{req_{M.p}} = \left\lceil \frac{6 \text{ ms - } 4 \text{ ms}}{4 \text{ ms}} \right\rceil = 1$$

The SystemC-AMS simulation output of the model from Figure 4.1 using a delay in port **B.out** of $\mathrm{D_{out}} = 1$ is shown in Figure 4.9. The details of the simulation's execution of the model are shown below.

```
============ MODULE A activated @ t_tdf = 0 s ============
 A @ t_tdf = 0 s: A.in (converter port) reading 0
 A @ t_tdf = 0 s: A.out writing 1
 A @ t_tdf = 2 ms: A.out writing 2
 A @ t_tdf = 4 ms: A.out writing 3
============ MODULE B activated @ t_tdf = 0 s ============
 B @ t_tdf = 0 s: B.in reading 1
 B @ t_tdf = 2 ms: B.in reading 2
 B @ t_tdf = 0 s : B.out (converter port) writing 3
============ MODULE A activated @ t_tdf = 6 ms ============
X @ t_de = 0 s : MODULE X activated
 X @ t_de = 0 s: X.out writing 5
Y @ t_de = 0 s : MODULE Y activated
 Y @ t_de = 0 s: Y.in reading 0
X @ t_de = 1 ms : MODULE X activated
 X @ t_de = 1 ms: X.out writing 5
Y @ t_de = 1 ms : MODULE Y activated
 Y @ t_de = 1 ms: Y.in reading 0
X @ t_de = 2 ms : MODULE X activated
 X @ t_de = 2 ms: X.out writing 5
Y @ t_de = 2 ms : MODULE Y activated
 Y @ t_de = 2 ms: Y.in reading 0
X @ t_de = 3 ms : MODULE X activated
 X @ t_de = 3 ms: X.out writing 5
Y @ t_de = 3 ms : MODULE Y activated
 Y @ t_de = 3 ms: Y.in reading 0
Y @ t_de = 4 ms : MODULE Y activated
 Y @ t_de = 4 ms: Y.in reading 3
X @ t_de = 4 ms : MODULE X activated
 X @ t_de = 4 ms: X.out writing 5
X @ t_de = 5 ms : MODULE X activated
 X @ t_de = 5 ms: X.out writing 5
Y @ t_de = 5 ms : MODULE Y activated
 Y @ t_de = 5 ms: Y.in reading 3
 A @ t_tdf = 6 ms: A.in (converter port) reading 5
 A @ t_tdf = 6 ms: A.out writing 1
```

(a)

```
 A @ t_tdf = 6 ms: A.out writing 1
 A @ t_tdf = 8 ms: A.out writing 2
 A @ t_tdf = 10 ms: A.out writing 3
============ MODULE B activated @ t_tdf = 4 ms ============
 B @ t_tdf = 4 ms: B.in reading 3
 B @ t_tdf = 6 ms: B.in reading 1
 B @ t_tdf = 4 ms : B.out (converter port) writing 4
============ MODULE B activated @ t_tdf = 8 ms ============
 B @ t_tdf = 8 ms: B.in reading 2
 B @ t_tdf = 10 ms: B.in reading 3
 B @ t_tdf = 8 ms : B.out (converter port) writing 5
X @ t_de = 6 ms : MODULE X activated
 X @ t_de = 6 ms: X.out writing 5
Y @ t_de = 6 ms : MODULE Y activated
 Y @ t_de = 6 ms: Y.in reading 3
X @ t_de = 7 ms : MODULE X activated
 X @ t_de = 7 ms: X.out writing 5
Y @ t_de = 7 ms : MODULE Y activated
 Y @ t_de = 7 ms: Y.in reading 3
X @ t_de = 8 ms : MODULE X activated
 X @ t_de = 8 ms: X.out writing 5
Y @ t_de = 8 ms : MODULE Y activated
 Y @ t_de = 8 ms: Y.in reading 4
X @ t_de = 9 ms : MODULE X activated
 X @ t_de = 9 ms: X.out writing 5
Y @ t_de = 9 ms : MODULE Y activated
 Y @ t_de = 9 ms: Y.in reading 4
X @ t_de = 10 ms : MODULE X activated
 X @ t_de = 10 ms: X.out writing 5
Y @ t_de = 10 ms : MODULE Y activated
 Y @ t_de = 10 ms: Y.in reading 4
X @ t_de = 11 ms : MODULE X activated
 X @ t_de = 11 ms: X.out writing 5
Y @ t_de = 11 ms : MODULE Y activated
 Y @ t_de = 11 ms: Y.in reading 4
============ MODULE A activated @ t_tdf = 12 ms ============
 A @ t_tdf = 12 ms: A.in (converter port) reading 5
```

(b)

Figure 4.9: SystemC-AMS simulation of model accessing input converter port before accessing output converter port using a delay to solve causality problems.

Table 4.5 shows how the t_{TDF} and t_{DE} advance when a synchronization operation occurs, according to each step of the below description. It also shows if Equation 4.1 holds or if a causality problem is generated.

1. At start of the simulation, t_{DE}, t_{TDF_A}, and t_{TDF_B} are at 0 ms. Because of the delay $\mathrm{D_{out}} = 1$ in the output converter port **B.out**, there is a sample already available at $t_{\mathrm{TDF}_{B.out}} = 0$ ms.

2. Module **A** activates at $t_{\mathrm{TDF}_A} = 0$ ms, but for module **A** to be executed, an access to the input converter port **A.in** occurs, generating a read synchronization operation. In consequence, the SystemC DE simulation kernel is executed and t_{DE} advances to 0 ms. The step is not shown in Figure 4.9, because when the access to the input converter port **A.in** happens, the t_{DE} is already equal to the $t_{\mathrm{TDF}_{A.in}}$.

Step	Sync. Type	t_{TDF} (ms)	t_{DE} (ms)	$t_{\mathrm{TDF}} \geq t_{\mathrm{DE}}$	New t_{DE} (ms)
1	–	$t_{\mathrm{TDF}_{B.out}} = 0$	0	–	–
2	Read	$t_{\mathrm{TDF}_{A.in}} = 0$	0	True	0
3	–	–	–	–	–
4	–	–	–	–	–
5	Write	$t_{\mathrm{TDF}_{B.out}} = 4$	0	True	0
6	Read	$t_{\mathrm{TDF}_{A.in}} = 6$	0	True	6
7	–	–	–	–	–
8	–	–	–	–	–
9	Write	$t_{\mathrm{TDF}_{B.out}} = 8$	6	True	6
10	–	–	–	–	–
11	Write	$t_{\mathrm{TDF}_{B.out}} = 12$	6	True	6
12	Periodic	$t_{\mathrm{TDF}} = 12$	6	True	12

Table 4.5: TDF and DE simulation time tracking for model accessing input converter port before accessing output converter port using a delay to solve causality problems.

3. Then, module A reads the available sample from the input converter port **A.in** at 0 ms, executes its internal functions and writes three samples into the output port **A.out** with timestamps of 0 ms, 2 ms and 4 ms respectively. This is shown at the upper part of Figure 4.9a.

4. Module **B** is activated at $t_{\mathrm{TDF}_B} = 0$ ms. As shown in Figure 4.9a, it reads two of the three available samples from the input port **B.in** at 0 ms and 2 ms respectively, executes its internal functions and writes one sample into the output converter port. But because a delay of $\mathrm{D_{out}} = 1$ was inserted in the output converter port **B.out**, this sample will actually be written to the DE module at 4 ms. This can be seen in Figure 4.9a, when the module **Y** reads a value of 0 from $t_{\mathrm{DE}} = 0$ ms to 3 ms, and then it starts reading the current sample from module **B** at $t_{\mathrm{DE}} = 4$ ms. Hence it is considered that the access to the output converter port occurs at 4 ms due to the action of the delay.

5. Since an access to the output converter port **B.out** occurred at 4 ms, a write synchronization operation is generated. Equation 4.1 holds since $t_{\mathrm{TDF}_{B.out}} = 4$ ms and $t_{\mathrm{DE}} = 0$ ms, and the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

6. According to the schedule, the next module to be executed is **A**. Since the timestep of the module is $\mathrm{Tm}_A = 6$ ms, it activates at $t_{\mathrm{TDF}_A} = 6$ ms as Figure 4.9a shows. But once again in order to be executed, an access to the input converter port **A.in** will occur at 6 ms and a read synchronization operation will be generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{A.in}} = 6$ ms and $t_{\mathrm{DE}} = 0$ ms. As Figure 4.9a shows, the SystemC DE simulation kernel is executed and t_{DE} advances until it reaches $t_{\mathrm{TDF}_{A.in}}$; that is $t_{\mathrm{DE}} = 6$ ms. Figure 4.9a shows that the **X** and **Y** modules execute up to $t_{\mathrm{DE}} = 5$ ms. After this, the t_{DE} advances to 6 ms and then the SystemC-AMS simulation kernel takes control.

7. Now module **A** reads the available sample from the input converter port **A.in** at 6 ms as shown at the bottom of Figure 4.9a, executes its internal functions and writes three samples into the output port **A.out** with timestamps of 6 ms, 8 ms and 10 ms respectively, as it is shown in the upper part of Figure 4.9b.

8. The next module to be activated according to the schedule is module **B**. Its timestep is $\mathrm{Tm}_B = 4$ ms, hence it is activated at $t_{\mathrm{TDF}_B} = 4$ ms. It reads two of the four available samples from the input port **B.in** at 4 ms and 6 ms respectively, as shown in Figure 4.9b. It executes its internal functions and tries to write one sample into the output converter port **B.out** at 8 ms (due to the action of the delay).

9. Since an access to the output converter port **B.out** occurred at 8 ms, a write synchronization operation is generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{B.out}} = 8$ ms and $t_{\mathrm{DE}} = 6$ ms, but the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

10. As there are still two available samples in the port **B.in** to be consumed, module **B** is activated again at $t_{\mathrm{TDF}_B} = 8$ ms as shown in Figure 4.9b. It reads the last two available samples from the input port **B.in** at 8 ms and 10 ms respectively, executes its internal functions and writes one sample into the output converter port **B.out** at 12 ms.

11. Since an access to the output converter port **B.out** occurred at 12 ms, a write synchronization operation is generated. Equation 4.1 still holds since $t_{\mathrm{TDF}_{B.out}} = 12$ ms and $t_{\mathrm{DE}} = 6$ ms, but the SystemC DE simulation kernel does not execute and t_{DE} does not advance.

12. At this point, one period of the schedule of the model has been executed. The last sample that is available in the output converter port **B.out** represents the delay sample corresponding to the next period of the execution of the model. A periodic synchronization operation is generated, and as Figure 4.9b shows, the SystemC DE simulation kernel finally executes and t_{DE} advances from 6 ms until it reaches the new t_{TDF} for the start of a new TDF cluster period; that is until $t_{\mathrm{DE}} = 12$ ms. At the bottom of Figure 4.9b, the execution of a new TDF cluster period at $t_{\mathrm{TDF}} = 12$ ms occurs. And at this point in time, the SystemC DE simulation kernel is interrupted.

4.3. Proposal for detection and avoidance of time synchronization issues

The SystemC-MDVP simulator can detect time synchronization issues between the DE and TDF MoCs by transforming the TDF clusters and their interaction with the DE domain into an equivalent timed-CPN model. But to do that, the SystemC code should be generated and executed in order to find any possible causality problems during the elaboration (pre-simulation) phase and provide the necessary delay suggestions to avoid these issues. Following TTool's philosophy of verification and validation of the models at the partitioning and software design levels, the synchronization between the DE and TDF MoCs should be validated before the SystemC-AMS code is generated. Therefore, another approach was explored.

Referring to the results of the analysis presented in Subsection 4.1.5, in the SystemC-AMS simulation kernel time synchronization issues between DE and TDF MoCs only occur when there is a read synchronization operation that advances the t_{DE} further than the t_{TDF} of the following write synchronization operation. The approach presented here is based on this fact.

Whenever there are multi-rate TDF clusters that interact with the DE domain using input/output converter ports, a time analysis including all the input and output converter ports is required. The interactions of each converter port of a TDF cluster with the DE domain cannot be considered independently, since they maintain a time relation with each TDF converter port and their DE counterparts. The time relation is based on the static schedule of the TDF Cluster which is pre-computed in the same way as the static schedule of an SDF MoC [25]. This means that once a static schedule has been found, the timestamps of the DE domain should be tracked after each execution of a TDF module that accesses a converter port. The following equations derived from [7, p. 51] show how to perform this DE timestamp tracking. As shown in Figure 4.10, where only the properties of converter ports are represented, Tm_{M} is the timestep of the TDF module **M**; Tp_{p} is the timestep of the TDF converter port **p** of module **M**; D_{p} is the delay associated to this converter port and R_{p} is the rate associated to the converter port.

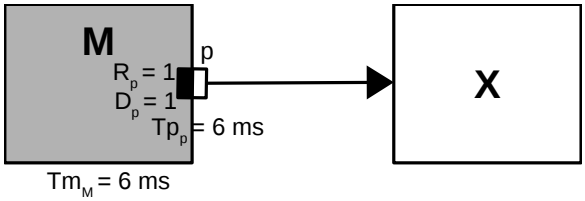


Figure 4.10: Parameters of the TDF module and its converter ports.

Here, j_{M} is the number of times that the TDF module **M** has been executed, considering that j_{M} is increased only when the number of samples indicated by the rates of the input and output ports have been consumed/produced and the module has finished executing. Finally, k represents the number of times that the converter port has been accessed within one activation of the module **M**—i.e. the sample number that has been produced or consumed.

Equation 4.3 shows how to calculate the DE timestamps for input converter ports. In other words, this equation determines how the t_{DE} advances when there is an access to an input converter port.

$$t_{\mathrm{DE}} = (j_{\mathrm{M}} \cdot Tm_{\mathrm{M}}) + ((k-1) \cdot Tp_{\mathrm{p}}) - D_{\mathrm{p}} \cdot Tp_{\mathrm{p}} \qquad k = [1...R_{\mathrm{p}}] \tag{4.3}$$

On the other hand, Equation 4.4 is used in order to determine if a future access to an output converter port will generate a causality problem. This equation determines the $t_{\mathrm{TDF}_{M.p}}$ of an output converter port **p** from a module **M**.

$$t_{\mathrm{TDF}_{M.p}} = (j_{\mathrm{M}} \cdot Tm_{\mathrm{M}}) + ((k-1) \cdot Tp_{\mathrm{p}}) + D_{\mathrm{p}} \cdot Tp_{\mathrm{p}} \qquad k = [1...R_{\mathrm{p}}] \tag{4.4}$$

As mentioned before, the time synchronization issues only occur when there is an access to an input converter port before an access to an output converter port. So in order to validate that Equation 4.1 always holds, only read synchronization operations before write synchronization operations will be analyzed. For this purpose and based on the static schedule for one complete TDF cluster period, every time each module is executed, the minimum timestamp $t_{\mathrm{TDF}_{M.p}}$ of all the accessed output converter ports of the current executed module should be greater or equal than the maximum timestamp (t_{DE}) of all

the accessed input converter ports of previously executed TDF modules in the schedule. This is shown in Equation 4.5

$$MIN\big(t_{\mathrm{TDF}_{M.p}}(m(o))\big) \geq MAX\big(t_{\mathrm{DE}}(n(i))\big) \qquad \begin{aligned} &m = current\_executed\_module, \\ &o = [1...\#out\_converter\_ports], \\ &n = [first\_executed\_module...m], \\ &i = [1...\#in\_converter\_ports] \end{aligned} \tag{4.5}$$

If Equation 4.5 does not hold, then a delay calculated from Equation 4.2 should be inserted in the corresponding output converter port. It is important to mention, that even if these equations are only applied for the case when there is a read synchronization operation before a write synchronization operation, the same equations will work in all the other possible scenarios. Only the parameters i and o from Equation 4.5 should be adapted to the specific scenario that is being validated.

```
1: procedure WALKTHROUGHSCHEDULE(static_schedule)
2:     max_t_DE ← 0
3:     j_M ← 0                                          ▷ For each module M
4:     for each module M in static_schedule do
5:         DETECTTIMESYNCISSUES(M.converterPorts, j_M, max_t_DE)
6:         j_M ← j_M + 1
7:     end for
8: end procedure
```

```
1: procedure DETECTTIMESYNCISSUES(converterPorts, j_M, max_t_DE)
2:     for each converterPort p in converterPorts do
3:         if p.origin = input then
4:             k ← R_p
5:             t_DE ← (j_M · Tm_M) + ((k − 1) · Tp_p) − D_p · Tp_p
6:             max_t_DE ← max(max_t_DE, t_DE)
7:         else if p.origin = output then
8:             k ← 1
9:             t_TDF_M.p ← (j_M · Tm_M) + ((k − 1) · Tp_p) + D_p · Tp_p
10:            if !(t_TDF_M.p ≥ max_t_DE) then
11:                D_req ← ⌈(max_t_DE − t_TDF_M.p)/Tp_p⌉
12:                CAUSALITY ERROR - STOP
13:            end if
14:        end if
15:    end for
16: end procedure
```

Listing 4.1: Algorithm to detect time synchronization issues.

The algorithm shown in Listing 4.1, makes use of the previously defined Equations to detect time synchronization issues between the TDF and DE MoCs and to suggest an appropriate port Delay that can solve these causality problems.

In the procedure `walkThroughSchedule` from lines 1-8, the global variable $\texttt{max\_t}_{\texttt{DE}}$ (initialized to 0 in line 2) will be used to store the maximum calculated t_{DE} from all the accessed input converter ports. The variable $\texttt{j}_{\texttt{M}}$ (number of times that module

M has been executed) is local to each module M and will be initialized to 0 for each module M of the model at line 3. In lines 4-7 the procedure will go over the pre-computed static schedule list, and for each module M in the schedule, it will call the `detectTimeSyncIssues` procedure, giving as parameters the list of all converter ports of the module M (`M.converterPorts`), and the variables $\texttt{j}_\texttt{M}$ and $\texttt{max\_t}_\texttt{DE}$. After this in line 6, it will increment $\texttt{j}_\texttt{M}$, meaning that the module M has been executed one more time.

The `detectTimeSyncIssues` procedure in lines 1-16 will check for time synchronization issues in the current scheduled module M. In lines 2-15, it will go over the list of `converterPorts`. In line 3, it will check if it is an input converter port. Note that in line 4 the variable `k` is set to the value of the port rate R_p. This is because according to Equation 4.5, we want to get the biggest value of t_DE for that input converter port. So in this case, it is useless to calculate the t_DE for previous samples (in case the port Rate is greater than 1). After this, it will calculate the t_DE in line 5 using Equation 4.3. Then in line 6, the computed t_DE is compared against the previously stored value $\texttt{max\_t}_\texttt{DE}$, to keep track of the maximum calculated t_DE from all the accessed input converter ports.

In line 7, it will check if the converter port is an output converter port. Notice that according to Equation 4.5, we are looking for the minimum $t_{\text{TDF}_{M.p}}$ of all the accessed output converter ports of the current executed module. For that reason, the variable `k` is set to 1 in line 8, since calculating bigger values of $t_{\text{TDF}_{M.p}}$ is useless in this case. Then, the $t_{\text{TDF}_{M.p}}$ will be calculated in line 9 using Equation 4.4.

Now that the minimum $t_{\text{TDF}_{M.p}}$ has been computed, Equation 4.5 can be applied. This is done in line 10. And as explained before, if this equation does not hold, it means that a causality problem is present and a delay should be inserted in that output converter port to solve the problem. This delay is computed in line 11 using Equation 4.2. After this, the execution of the program is stopped because a time synchronization issue was found (line 12).

Step	Executed Module/ port (M.p)	j_M	Tm_M (ms)	Tp_p (ms)	D_p	t_DE (ms) (Eq. 4.3)	max_t_DE (ms)	$t_{\text{TDF}_{M.p}}$ (ms) (Eq. 4.4)	Causality Check (Eq. 4.5)
1	A.in	0	6	6	0	0	0	–	–
2	A.in	1	6	6	0	0	0	–	–
3	B.out	0	4	4	0	–	0	0	True
4	B.out	1	4	4	0	–	0	0	–
5	A.in	1	6	6	0	6	6	–	–
6	A.in	2	6	6	0	6	6	–	–
7	B.out	1	4	4	0	–	6	4	**False**

Table 4.6: Execution of model without delay and computation of DE and TDF simulation times.

Using the same model from Figure 4.1, the algorithm from Listing 4.1 has been applied during the execution of the model's simulation, in order to verify that the computed times are the same and that the synchronization issues are detected as in Sections 4.1 and 4.2. Table 4.6 shows all the parameters to calculate the t_DE, t_TDF and the detection of synchronization issues. Since the rates (R_p) of the converter ports of this model are equal to 1, the parameter k will always have a value of 1 and it is not shown in the

table. Below, a description of the execution and calculation of the simulation times of the model is presented. Remember that the static schedule of this model is **ABABB**.

1. Starting from the `walkThroughSchedule` procedure, module A is executed for the first time ($j_{\mathrm{A}} = 0$) according to the schedule. Procedure `detectTimeSyncIssues` is called. Module A has only one converter port, which is an input converter port. Since $R_{\mathrm{in}} = 1$, k will be set up to 1 as mentioned before. The t_{DE} will be calculated and the max_t_{DE} will be chosen as shown in Table 4.6. Since there is no access to an output converter port, the causality check will not be performed.

2. There are no more converter ports, so the procedure `detectTimeSyncIssues` finishes and j_{A} is incremented.

3. Next, module B is executed for the first time ($j_{\mathrm{B}} = 0$) according to the schedule. Procedure `detectTimeSyncIssues` is called. Module B has only one converter port, which is an output converter port. Hence k is set up to 1, and $t_{\mathrm{TDF}_{B.out}}$ is calculated. Then the causality check using Equation 4.5 is performed. In this case $max\_t_{\mathrm{DE}} = 0$ ms and $t_{\mathrm{TDF}_{B.out}} = 0$ ms so there is no causality problem.

4. Module B has no more converter ports, so `detectTimeSyncIssues` finishes and j_{B} is incremented.

5. Now, module A is executed for the second time ($j_{\mathrm{A}} = 1$). The procedure `detectTimeSyncIssues` is called. For its only input converter port the t_{DE} is calculated and max_t_{DE} is chosen.

6. There are no more converter ports, so the procedure `detectTimeSyncIssues` finishes and j_{A} is incremented.

7. Then, module B is executed for the second time ($j_{\mathrm{B}} = 1$). The procedure `detectTimeSyncIssues` is called. For the output converter port, the $t_{\mathrm{TDF}_{B.out}}$ is calculated. As it can be seen in Table 4.6, since the max_t_{DE} of all previously accessed input converter ports is 6 ms and $t_{\mathrm{TDF}_{B.out}} = 4$ ms, Equation 4.5 does not hold and a causality problem is generated. This is the same behavior that was analyzed in Section 4.1. The minimum required delay is calculated as before using Equation 4.2, that is $D_{req_{B.out}} = 1$, and the execution is interrupted.

Now we can apply the same algorithm in order to verify that the synchronization issues are solved when this new delay is inserted. Table 4.7 shows how the simulation times are calculated using this new delay as well as the causality check. Below, the description of the execution of the model with the new delay added is presented.

1. Starting from the `walkThroughSchedule` procedure, module A is executed for the first time ($j_{\mathrm{A}} = 0$) according to the schedule. Procedure `detectTimeSyncIssues` is called. Module A has only one converter port, which is an input converter port. Since $R_{\mathrm{in}} = 1$, k will be set up to 1 as mentioned before. The t_{DE} will be calculated and the max_t_{DE} will be chosen as shown in Table 4.7. Since there is no access to an output converter port, the causality check will not be performed.

2. There are no more converter ports, so the procedure `detectTimeSyncIssues` finishes and j_{A} is incremented.

Step	Executed Module/ port (M.p)	j_{M}	Tm_{M} (ms)	Tp_{p} (ms)	D_{p}	t_{DE} (ms) (Eq. 4.3)	max_t_{DE} (ms)	$t_{\mathrm{TDF}_{M.p}}$ (ms) (Eq. 4.4)	Causality Check (Eq. 4.5)
1	A.in	0	6	6	0	0	0	–	–
2	A.in	1	6	6	0	0	0	–	–
3	B.out	0	4	4	1	–	0	4	True
4	B.out	1	4	4	1	–	0	4	–
5	A.in	1	6	6	0	6	6	–	–
6	A.in	2	6	6	0	6	6	–	–
7	B.out	1	4	4	1	–	6	8	True
8	B.out	2	4	4	1	–	6	8	True
9	B.out	2	4	4	1	–	6	12	True
10	B.out	3	4	4	1	–	6	12	True

Table 4.7: Execution of model with delay and computation of DE and TDF simulation times.

3. Next, module B is executed for the first time ($j_{\mathrm{B}} = 0$) according to the schedule. Procedure `detectTimeSyncIssues` is called. Since module B has only one output converter port, the $t_{\mathrm{TDF}_{B.out}}$ is calculated using the new delay $D_{out} = 1$ as shown in Table 4.7. Then the causality check using Equation 4.5 is performed. In this case $max\_t_{\mathrm{DE}} = 0$ ms and $t_{\mathrm{TDF}_{B.out}} = 4$ ms so there is no causality problem.

4. Module B has no more converter ports, so `detectTimeSyncIssues` finishes and j_{B} is incremented.

5. Now, module A is executed for the second time ($j_{\mathrm{A}} = 1$). The procedure `detectTimeSyncIssues` is called. For its only input converter port the t_{DE} is calculated and max_t_{DE} is chosen.

6. There are no more converter ports, so the procedure `detectTimeSyncIssues` finishes and j_{A} is incremented.

7. Then, module B is executed for the second time ($j_{\mathrm{B}} = 1$). The procedure `detectTimeSyncIssues` is called. For the output converter port, the $t_{\mathrm{TDF}_{B.out}}$ is calculated. As it can be seen in Table 4.7, since the max_t_{DE} of all previously accessed input converter ports is 6 ms and $t_{\mathrm{TDF}_{B.out}} = 8$ ms, Equation 4.5 holds and there is no causality problem.

8. Module B has no more converter ports, so `detectTimeSyncIssues` finishes and j_{B} is incremented.

9. According to the schedule, module B needs to be executed one last time ($j_{\mathrm{B}} = 2$). The procedure `detectTimeSyncIssues` is called. For the output converter port, the $t_{\mathrm{TDF}_{B.out}}$ is calculated. Once again, since $max\_t_{\mathrm{DE}} = 6$ ms and $t_{\mathrm{TDF}_{B.out}} = 12$ ms, Equation 4.5 holds and there is no causality problem.

10. Finally, module B has no more converter ports, so `detectTimeSyncIssues` finishes and j_{B} is incremented.

As it occurred in Section 4.2, the synchronization issues were avoided with the introduction of the delay.

With this approach, analysis at the design level can be performed, which allows the detection of time synchronization issues before the SystemC-AMS code of a model is generated, and provides suggestions on the needed delay modifications to avoid these synchronization issues.

This solution was chosen for the integration and synchronization of the SystemC-AMS and SoCLib components in TTool due to its easiness of implementation. In Chapter 6, the simulation results of different models are shown, to prove the correct behavior of this approach by comparing it with the SystemC-AMS and SystemC-MDVP simulators.

5. Integration of SystemC-AMS and SoCLib modules into TTool

5.1. Integration decisions to augment the graphical interface of TTool

During phase 2 of the project and before integration of the SoCLib and SystemC-AMS TDF modules in TTool phase started, some decisions were taken concerning the way the graphical interface must be augmented, which were implemented as part of the work of [4]. Below is a description of these changes to the graphical interface.

Since the actual code generation of the platform is done from the Deployment Diagrams, the SystemC-AMS TDF clusters should be visible here. The Deployment Diagram of TTool was thus augmented with a representation of each TDF cluster. Figure 5.1 shows the new SystemC-AMS Cluster block (gray) that was created in order to offer a graphical interface to initiate with the integration between the SystemC-AMS TDF modules and the SoCLib modules from the Deployment Diagram. When this block is inserted, it allows only to modify its name, which should be the same name as the TDF cluster created in the SystemC-AMS Component Diagram panel. When this block is double-clicked, a panel containing a corresponding SystemC-AMS Component Diagram will open.

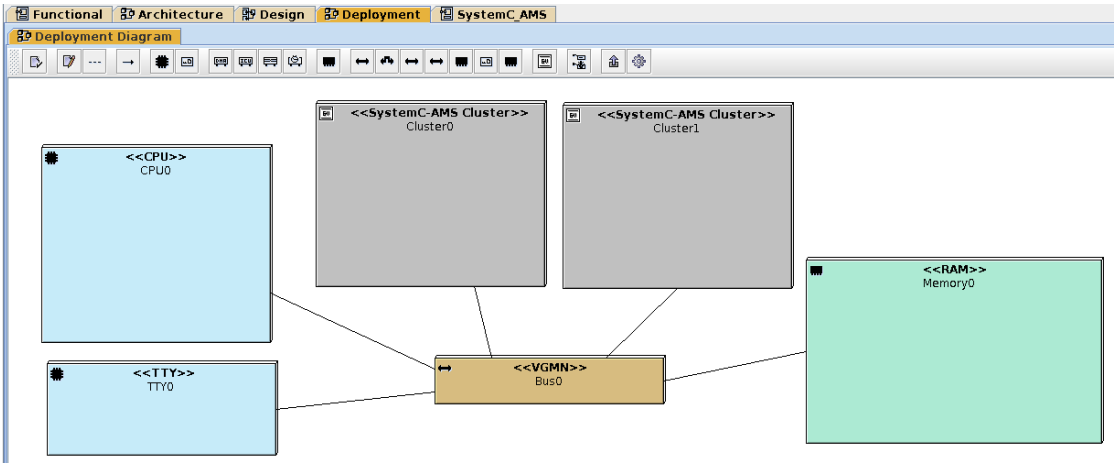


Figure 5.1: Deployment Diagram with two SystemC-AMS Cluster blocks.

As shown in Figure 5.1, several SystemC-AMS Cluster blocks can be created and they can be connected to the SoCLib interconnect component which uses the VCI protocol. As explained in Section 3.1, a generic adaptor component (GPIO2VCI) was developed as a VCI target, which works as an interface between the SystemC-AMS and the SoCLib interconnect components. Since the SystemC-AMS TDF cluster will be connected to the GPIO2VCI component, the graphical interface of the SystemC-AMS Component Diagram was augmented too.

Figure 5.2 shows a new block called `blockGPIO2VCI`, that can be created in the SystemC-AMS Component Diagram panel. This block does not have any attributes and it must be created outside of the TDF cluster. It only allows to insert DE ports, which will be used to connect the `blockGPIO2VCI` to a TDF or DE module. Note that a dotted connector is used to show that the module is connected to the `blockGPIO2VCI`. This connector and the new `blockGPIO2VCI` will be used for the later integration development. Finally, a new

data type `sc_uint<32>` was introduced in the list of data types to choose from the DE and TDF converter ports. This data type is needed to allow the TDF or DE modules to connect to the GPIO2VCI component, because as it was explained in Section 3.1, the ports of the GPIO2VCI components have a data type that is internally defined as `sc_uint<32>`.

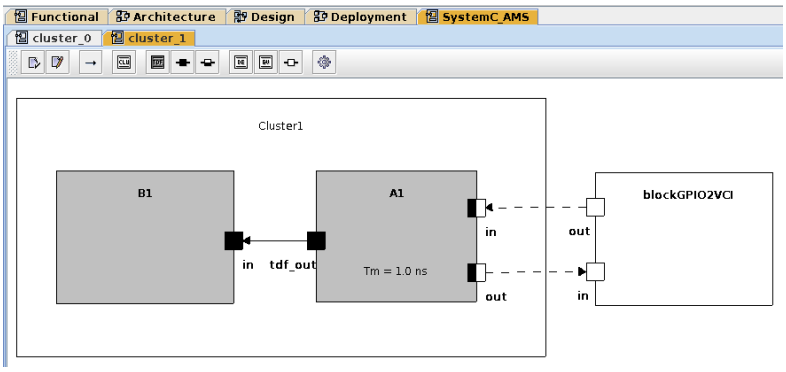


Figure 5.2: `blockGPIO2VCI` in the SystemC-AMS Component Diagram for Cluster1.

5.2. Integration of SystemC-AMS and the SoCLib modules in TTool

On one side and as explained before, the graphical interface of TTool was extended to allow the possibility of creating abstractions of AMS hardware components into a SystemC-AMS Component Diagram, which allows the generation of SystemC-AMS virtual prototypes of the AMS models [4]. On the other side, the abstraction of digital hardware components into a Deployment Diagram has been addressed [3], which allows the generation of SoCLib (SystemC) virtual prototypes of these digital hardware models. Now, the integration of TTool's TDF models with its SoCLib models is implemented in this phase.

The main idea for the integration is to use the topcell file generated from the Deployment Diagram to include the SystemC-AMS TDF clusters created in the SystemC-AMS Component Diagrams. For this, the first step is to be able to compile the topcell file while including the SystemC-AMS libraries. As it was explained in Section 3.3, the SoCLib platform is designed to work only with SystemC modules, but its configuration files can be modified to include the SystemC-AMS libraries. The SoCLib libraries in TTool are configured at the user home directory level, using a `global.conf` file that is located in the following path `$HOME/.soclib/global.conf`.

Listing A.6 from the Appendix shows the modified `global.conf` file. First, the SystemC version in the configuration file needed to be updated to use version 2.3.1 (line 24), since SystemC-AMS-2.1 does not work with SystemC-2.2.0, which was originally being used. In lines 29-41, the SystemC-AMS libraries, configuration flags and paths are included. This configuration was added in the libraries section of the build environment in line 50.

Once the generated topcell from the Deployment Diagram can be compiled including the SystemC-AMS libraries, the next step is to include the GPIO2VCI component into the SoCLib components directory. It was decided to create a new `gpio2vci/` directory under `$HOME/TTool/MPSoC/soclib/soclib/module/connectivity_component/`. Since the GPIO2VCI component is implemented as a CABA component, its definition, im-

plementation and Metadata files were created under the `caba/` directory following the SoCLib components structure, as it is described below.

Under the `connectivity_component/` directory, the GPIO2VCI component's definition file `gpio2vci.h` shown in Listing A.1, the GPIO2VCI component's implementation file `gpio2vci.cpp` shown in Listing A.2, and the Metadata file `gpio2vci.sd` shown in Listing A.3, were created and added to their corresponding directories.

In order to integrate the TDF modules into the SoCLib topcell, a cluster file that instantiates all the TDF modules of a cluster needs to be generated. In this way, the clusters can be included and instantiated directly in the `top.cc` topcell from the SoCLib modules, without having to instantiate each of the modules of an AMS cluster individually. Then each TDF cluster can be connected to the SoCLib interconnect using the GPIO2VCI component.

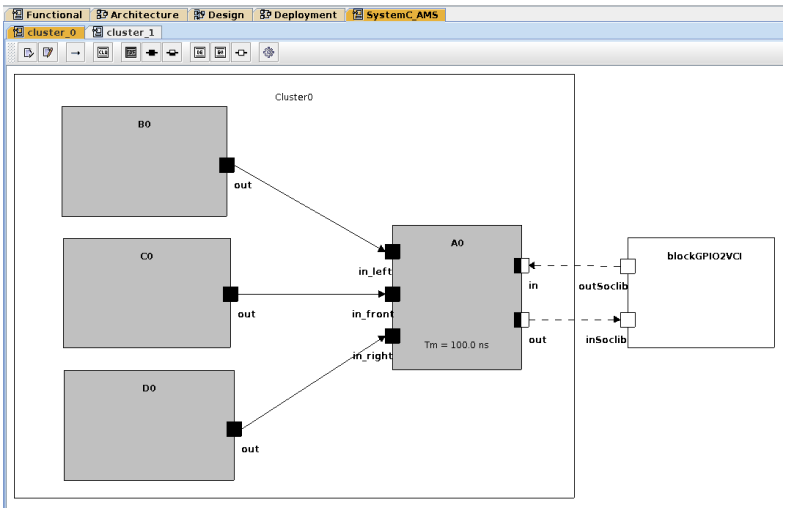


Figure 5.3: SystemC-AMS Cluster0 from the model shown in Figure 5.1.

The model presented in Figure 5.1 which contains two SystemC-AMS Cluster blocks (shown in Figures 5.2 and 5.3) will be used in the following descriptions to exemplify the generation of the SystemC-AMS modules and clusters code, as well as for the generation of the embedded software code. The model consists of the two SystemC-AMS Cluster blocks (gray), one CPU, one RAM memory, and one TTY console. All the components are connected via a VGMN interconnect component.

The TDF cluster `Cluster0` shown in Figure 5.3, consists of three TDF modules (**B0**, **C0** and **D0**) all connected to one TDF module (**A0**). Module **A0** is connected to the `blockGPIO2VCI` component. These TDF cluster does not replicate any particular behavior. It is used to explain the SystemC-AMS generated code, focusing on the attributes of the modules and the cluster.

The TDF cluster `Cluster1` shown in Figure 5.2, consists of two TDF modules (**A1** and **B1**). They replicate the Sine Source and Sink model used in Sections 3.2 and 3.3. Module **A1** contains the SystemC-AMS code for the Sine Source, and module **B1** has the SystemC-AMS code for the Sink.

The following `.java` files were created. They are responsible for generating the SystemC-AMS modules and clusters code.

PrimitiveCode.java: This file generates the SystemC-AMS code of the TDF/DE modules created under the SystemC-AMS Component Diagram. The generated code is stored as a header (`.h`) file for each module using the name of the block from the diagram as `blockname_tdf.h`. Each module is created as a C++ class taking the name from the block of the diagram and inheriting from the SystemC-AMS `sca_tdf::sca_module` class. The module includes all its ports and attributes defined in the SystemC-AMS Component Diagram. The SystemC-AMS code for the `processing()` function, that is created manually in the SystemC-AMS Component Diagram is also included in this header file. The same code developed in [4] was used for this purpose.

Listing 5.1 shows the class and the ports of file `A0_tdf.h` generated from the TDF module **A0** from Cluster0 shown in Figure 5.3. As it can be seen in the figure, this module has 5 ports which are instantiated in the SystemC-AMS code. Note that the data type of the ports that are connected to the `blockGPIO2VCI` in the model is `sc_uint<32>`.

```
class A0 : public sca_tdf::sca_module {

public:
        sca_tdf::sca_in< int > in_left;
        sca_tdf::sca_in< int > in_front;
        sca_tdf::sca_in< int > in_right;
        sca_tdf::sca_de::sca_out< sc_uint<32> > out;
        sca_tdf::sca_de::sca_in< sc_uint<32> > in;
```

Listing 5.1: Class and ports of the SystemC-AMS code generated for module **A0**.

ClusterCode.java: This file was created to generate the cluster code that instantiates the AMS modules from a TDF cluster. For each TDF cluster, the cluster code is being generated as a header (`.h`) file. The name of the file takes the name of the TDF cluster from the SystemC-AMS Component Diagram as `clustername_tdf.h`. Each cluster is created as a templated C++ class named after the cluster's name, and inheriting from the SystemC `sc_core::sc_module` class, as shown in Listing 5.2. This means that the cluster is created as a SystemC module. The template parameter that the class receives, is related to the VCI parameters that are defined in the `top.cc` topcell from the SoCLib modules.

Inside the cluster's class, the necessary signals to interconnect the different TDF/DE modules are declared as `sc_core::sc_signal`. As it is presented in Listing 5.2, they take their name from the signal name defined in the SystemC-AMS Component Diagram, or using a default name `sig_#` if no name was defined before; being `#` the number of signal that is declared.

Then each TDF/DE module from the cluster is instantiated using the name of the block in the form `blockname_#`, being `#` the number of block being instantiated, as it can be seen in Listing 5.2. Two default ports are generated for each cluster: `in_ams` which is a `sc_in` port; and `out_ams` which is a `sc_out` port. Both ports are of type `vci_param::data_t` (internally defined as `sc_uint<32>`), and they will be used to connect later to the GPIO2VCI component.

Inside the constructor `SC_CTOR` of the module, shown in Listing 5.3, the Net-List of the TDF cluster is created, using the previously declared signals to interconnect the TDF/DE modules through their respective ports. As it was depicted in Figure 5.2, modules **B0**, **C0** and **D0** are connected to module **A0** using signals `sig_0`, `sig_1` and

```
template <typename vci_param>
class Cluster0 : public sc_core::sc_module {
        sca_tdf::sca_signal<int> sig_0;
        sca_tdf::sca_signal<int> sig_1;
        sca_tdf::sca_signal<int> sig_2;
    // Instantiate cluster's modules.
        A0 A0_0;
        C0 C0_1;
        B0 B0_2;
        D0 D0_3;
public:
        sc_in< typename vci_param::data_t > in_ams;
        sc_out< typename vci_param::data_t > out_ams;
```

Listing 5.2: Class, ports and modules of the SystemC-AMS code generated for Cluster0.

```
template <typename vci_param>
SC_CTOR(Cluster0) :
...
{
        A0_0.in_left(sig_2);
        A0_0.in_front(sig_1);
        A0_0.in_right(sig_0);
        A0_0.out(out_ams);
        A0_0.in(in_ams);
        C0_1.out(sig_1);
        B0_2.out(sig_2);
        D0_3.out(sig_0);
}
```

Listing 5.3: Constructor `SC_CTOR` of the SystemC-AMS code generated for Cluster0.

`sig_2`. It is important to notice that the ports `A0_0.in` and `A0_0.out` that are connected to the `blockGPIO2VCI` in the SystemC-AMS Component Diagram, will be connected here directly to the `in_ams` and `out_ams` ports of the cluster. In this way, the cluster module will be the one that will be connected later to the GPIO2VCI component once it is instantiated in the `top.cc` topcell from the SoCLib modules.

Finally, a function invoking the SystemC-AMS tracing function `sca_trace` is implemented. This function will allow to create trace files using the signals of the AMS components of the cluster. The function is named using the cluster's name in the form `trace_clustername` as shown in Listing 5.4. This function can be called later from the `top.cc` topcell of the SoCLib modules, if tracing is required.

```
void trace_Cluster0(sca_util::sca_trace_file* tf) {
                sca_trace(tf, sig_0, "sig_0");
                sca_trace(tf, sig_1, "sig_1");
                sca_trace(tf, sig_2, "sig_2");
        }
```

Listing 5.4: Trace function of the SystemC-AMS code generated for Cluster0.

Header.java: This file is used to create the `#include` list that is added to both, the modules and clusters generated files. This code was slightly modified from the original version of [4], to adapt to the new files naming conventions.

TopCellGeneratorCluster.java: This file is the one in charge of creating the generated SystemC AMS code for the clusters and modules and saving them into their respective

directories. This file was also slightly modified from the original version of [4] to create the files for this phase.

Now that the SystemC-AMS code for the modules and the clusters is being generated, the clusters' code needs to be included and instantiated in the topcell (`top.cc`) generated from the Deployment Diagram. For this task, the following `.java` files were modified. These files generate the `top.cc` topcell code.

Header.java: It generates the `#include` files list in the `top.cc` topcell. If a SystemC-AMS Cluster block was added to the Deployment Diagram, then the `top.cc` topcell will include the GPIO2VCI component definition file `gpio2vci.h` and the TDF cluster file `clustername_tdf.h` for each cluster that was added to the Deployment Diagram. This is shown in Listing 5.5, where the generated files from the two clusters of the model shown in Figure 5.1 are being included along with the GPIO2VCI component.

```
#include "gpio2vci.h"
#include "Cluster0_tdf.h"
#include "Cluster1_tdf.h"
```

Listing 5.5: `#include` list of the `top.cc` topcell.

MappingTable.java: This file generates the mapping table and adds segments to it, including all the necessary parameters that the mapping table requires. For each TDF cluster that was created, it will add one segment to the mapping table for the GPIO2VCI component assigned to each cluster. The segments have the name `gpio2vci#`, being `#` the number of TDF cluster being added. The assigned addresses start from address `0xc0200000` and increase the address by `0x1000000` for every new segment that is added. Up to 16 TDF clusters can be handled by the mapping table, being the last possible assigned address `0xcf200000`. This address range was chosen, being an address that does not clash with any of the other possible addresses that can be assigned to other SoCLib components. A segment size of `0x10` was chosen, to give some freedom for the internal addresses that may be used within the GPIO2VCI component. Finally, the component target number is added as the fourth parameter as `IntTab(#)`. The Segments corresponding to the two clusters of the model being analyzed are shown in Listing 5.6. Note that the addresses `0xc0200000` and `0xc1200000` were asigned to each segment.

```
maptab.add(Segment("gpio2vci0" , 0xc0200000, 0x00000010, IntTab(12), false));
maptab.add(Segment("gpio2vci1" , 0xc1200000, 0x00000010, IntTab(13), false));
```

Listing 5.6: Segments of the mapping table of the `top.cc` topcell.

Declaration.java: The instantiation of the components is done here. Since the TDF clusters will be handled as VCI target components, the SoCLib interconnect components `VciVgsb` and `VciVgmn`, should update their internal parameters for the number of target components of the model, adding the number of TDF clusters that were created. This is shown in line 1 of Listing 5.7, where the `VciVgmn` component receives a value of 14 indicating that it is handling 14 target components. Then for each TDF cluster that was created, an instance of the `Gpio2Vci` component will be created, named in the form `gpio2vci#`, `#` being the number of TDF cluster assigned to the GPIO2VCI component This is shown in lines 3 and 6 of Listing 5.7. Also an instance of the TDF cluster component will be created, named as the TDF cluster name, as shown in lines 4 and 7.

```
soclib::caba::VciVgmn<vci_param> vgmn("Bus0" , maptab, 5,14,10,8);

caba::Gpio2Vci<vci_param> gpio2vci0("gpio2vci0", IntTab(12), maptab);
Cluster0<vci_param> Cluster0_0("Cluster0_0");

caba::Gpio2Vci<vci_param> gpio2vci1("gpio2vci1", IntTab(13), maptab);
Cluster1<vci_param> Cluster1_1("Cluster1_1");
```

Listing 5.7: Instantiation of clusters and GPIO2VCI component of the `top.cc` topcell.

Signal.java: This file creates the necessary signals for the model. For each TDF cluster that was created, three different signals will be instantiated. One `VciSignals` named `signal_vci_gpio2vci#`. One `sc_signal<vci_param::data_t>` named `signal_to_ams#`. And one `sc_signal<vci_param::data_t>` named `signal_from_ams#`. Remember that the `vci_param::data_t` data type is internally defined as `sc_uint<32>`, as explained in Section 3.1. For all the signals, `#` is the number of TDF cluster assigned to the GPIO2VCI component. The signals instantiated for the two clusters of the model are shown in Listing 5.8.

```
soclib::caba::VciSignals<vci_param> signal_vci_gpio2vci0("signal_vci_gpio2vci0");
sc_signal< vci_param::data_t > signal_to_ams0("signal_to_ams0");
sc_signal< vci_param::data_t > signal_from_ams0("signal_from_ams0");

soclib::caba::VciSignals<vci_param> signal_vci_gpio2vci1("signal_vci_gpio2vci1");
sc_signal< vci_param::data_t > signal_to_ams1("signal_to_ams1");
sc_signal< vci_param::data_t > signal_from_ams1("signal_from_ams1");
```

Listing 5.8: Signals of the `top.cc` topcell.

```
    gpio2vci0.p_clk(signal_clk);
    gpio2vci0.p_resetn(signal_resetn);
    gpio2vci0.p_vci(signal_vci_gpio2vci0);
    gpio2vci0.p_wdata_ams(signal_to_ams0);
    gpio2vci0.p_rdata_ams(signal_from_ams0);

    Cluster0_0.in_ams(signal_to_ams0);
    Cluster0_0.out_ams(signal_from_ams0);

    vgmn.p_to_target[12](signal_vci_gpio2vci0);

    gpio2vci1.p_clk(signal_clk);
    gpio2vci1.p_resetn(signal_resetn);
    gpio2vci1.p_vci(signal_vci_gpio2vci1);
    gpio2vci1.p_wdata_ams(signal_to_ams1);
    gpio2vci1.p_rdata_ams(signal_from_ams1);

    Cluster1_1.in_ams(signal_to_ams1);
    Cluster1_1.out_ams(signal_from_ams1);

    vgmn.p_to_target[13](signal_vci_gpio2vci1);
```

Listing 5.9: Net-List of the `top.cc` topcell.

Netlist.java: The Net-List that interconnects all the components is created here. For each created TDF cluster, the following interconnections will be generated, as shown in the generated Net-List from Listing 5.9. For the `gpio2vci0` component, the VCI target port `p_vci` is connected through the `signal_vci_gpio2vci0` signal to the port-to-VCI-target `p_to_target[12]` of the SoCLib Interconnect component, in this case

to a VGMN component `vgmn`. The port `p_wdata_ams` is connected through the signal `signal_to_ams0` to the input port `in_ams` of the TDF cluster component `Cluster0_0`. The port `p_rdata_ams` is connected through the signal `signal_from_ams0` to the output port `out_ams` of the `Cluster0_0`. In this way, the GPIO2VCI component connects the TDF clusters to the Interconnect component of the MPSoC. A similar interconnection is done for the second cluster `Cluster1_1` of the model.

Finally, the `Netlist.java` file offers the possibility to generate the code to create trace files. Listing 5.10 shows how tracing is handled for the SystemC-AMS components. In line 1, an `sca_util::sca_trace_file` object is created, and a tabular trace file is created with a given name. Then in lines 2 and 3 the signals that connect the GPIO2VCI component to the TDF cluster are added to the trace object. Finally in line 4, the tracing function created in the cluster's SystemC-AMS code (See Listing 5.4) is called and the trace object is sent as parameter. A similar thing happens for the second cluster in lines 6-8. Finally in line 10, the trace file is closed.

```
1  sca_util::sca_trace_file *tfp = sca_util::sca_create_tabular_trace_file("my_trace_analog");
2  sca_util::sca_trace(tfp,signal_to_ams0,"signal_to_ams0");
3  sca_util::sca_trace(tfp,signal_from_ams0,"signal_from_ams0");
4  Cluster0_0.trace_Cluster0(tfp);
5
6  sca_util::sca_trace(tfp,signal_to_ams1,"signal_to_ams1");
7  sca_util::sca_trace(tfp,signal_from_ams1,"signal_from_ams1");
8  Cluster1_1.trace_Cluster1(tfp);
9  ...
10 sca_util::sca_close_tabular_trace_file(tfp);
```

Listing 5.10: Tracing for the AMS components of the `top.cc` topcell.

It is important to mention that the tracing functionality can be enabled from the TTool graphical interface, but currently tracing is not yet fully functional for the SoCLib modules. Still, tracing for the SystemC signals of the GPIO2VCI component was added to the code, for when it becomes fully functional. This is depicted in Listing 5.11.

```
1  sc_trace(tf,signal_vci_gpio2vci0,"signal_vci_gpio2vci0");
2  sc_trace(tf,signal_to_ams0,"signal_to_ams0");
3  sc_trace(tf,signal_from_ams0,"signal_from_ams0");
4  sc_trace(tf,signal_vci_gpio2vci1,"signal_vci_gpio2vci1");
5  sc_trace(tf,signal_to_ams1,"signal_to_ams1");
6  sc_trace(tf,signal_from_ams1,"signal_from_ams1");
```

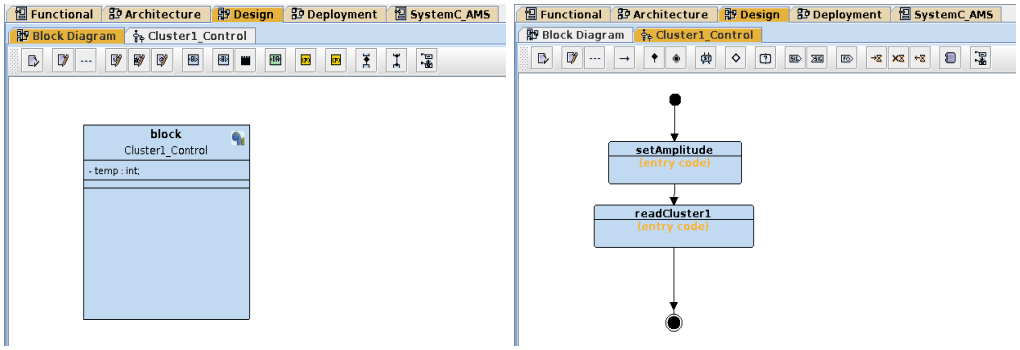
Listing 5.11: Tracing for the GPIO2VCI SystemC signals of the `top.cc` topcell.

PlatformInfo.java: This file generates the Platform Description File `platform_desc` similar to the one described in Section 3.3. In this case if any TDF cluster was created in the Deployment Diagram, then it will add an entry to the file to use the GPIO2VCI component, similar to the one shown in line 9 in Listing A.4 from the Appendix.

At this point, the SystemC-AMS code for the TDF modules and TDF clusters is being generated. The TDF clusters are being instantiated in the `top.cc` topcell of the SoCLib modules and connected to the SoCLib interconnect component using the GPIO2VCI component. Now, the software that will be loaded to the virtual prototype to communicate to the AMS modules needs to be generated.

The model of the TDF cluster `Cluster1` from Figure 5.2, which is a Sine Source (module **A1**) and Sink (module **B1**) model similar to the one used in Sections 3.2 and 3.3 will be used here. The idea is to create a software program at the software design level, that will communicate with the TDF cluster `Cluster1`. In Figure 5.4, the two diagrams that are used here are showed.

In the Block Diagram, the tasks of the components of a system can be represented by Blocks. As it is shown in Figure 5.4a, one Block `Cluster1_Control` has been created to represent the software tasks that will be used to communicate with the TDF cluster `Cluster1`. In the Software Design Diagrams, when a Block is created, automatically a new panel with the name of the Block is created. This panel allows to design the software of the specific task through a State Machine Diagram, as it can be seen in Figure 5.4b. In this case inside the `Cluster1_Control` State Machine Diagram, the first state block `setAmplitude` will set up the amplitude of the Sine Source module (module **A1**), and the second state block `readCluster1` will read the values generated by this module.



(a) **(b)**

Figure 5.4: Block Diagram and State Diagram for `Cluster1`.

Normally, the generated code from the state machine is correct-by-construction. But also, inside the state blocks of the State Machine Diagram, TTool allows to write C code manually, which later will be added to the generated software of the model. As it was shown in Sections 3.2 and 3.3, the communication with the AMS components through the GPIO2VCI component is done by writing to and reading from the assigned memory addresses of the GPIO2VCI component. So when the software needs to communicate with the AMS components, the C code that is written in the state blocks of the State Machine Diagram needs to specify these memory addresses. To facilitate the creation of the embedded software at this level, write and read functions were created to allow communication with the different TDF clusters.

To make this functions available to the cross-compiler, the first step was to create a new library `libsyscams` under `$HOME/TTool/MPSoC/mutekh/libsyscams/` which will contain the functions for the communication with the GPIO2VCI component. The following files were created inside this directory.

gpio2vci_iface.h: It defines the `write_gpio2vci` and `read_gpio2vci` functions as shown in Listing 5.12. The `write_gpio2vci` function receives the `data` parameter which represents the 32 bits of data that will be sent to the GPIO2VCI component. The parameter `name` is the name of the SystemC-AMS TDF cluster to which the data will be sent.

The `read_gpio2vci` function receives the `name` which is the name of the SystemC-AMS TDF cluster from which the data will be read.

```
#ifndef GPIO2VCI_IFACE_H
#define GPIO2VCI_IFACE_H

#include "gpio2vci_address.h"

void write_gpio2vci(int data, char name[]);

int read_gpio2vci(char name[]);

#endif //GPIO2VCI_IFACE_H
```

Listing 5.12: `gpio2vci_iface.h` definition file from the `libsyscams` library.

gpio2vci_iface.c: In the file shown in Listing 5.13 the implementation of these functions is given. The `write_gpio2vci` function uses a pointer created in line 4, in which the address from the GPIO2VCI component connected to the TDF cluster will be loaded. In line 5 the address is obtained by calling the `get_address` function which will be explained below. Then in line 6, the `data` is sent to that address. In a similar way, the `read_gpio2vci` function uses a pointer to read the values from the address of the GPIO2VCI component. In line 11 the address is loaded to the pointer by calling the `get_address` function, using the `name` of the TDF cluster. Then in line 12 the value read from that address is returned.

```
1  #include "gpio2vci_iface.h"
2
3  void write_gpio2vci(int data, char name[]) {
4      int * wr_ptr;
5      wr_ptr = (int*)get_address(name);
6      *wr_ptr = data;
7  }
8
9  int read_gpio2vci(char name[]) {
10     int * rd_ptr;
11     rd_ptr = (int*)(get_address(name)+4);
12     return *rd_ptr;
13 }
```

Listing 5.13: `gpio2vci_iface.h` implementation file from the `libsyscams` library.

gpio2vci_address.h: The file shown in Listing 5.14 defines the `get_address` function, which as seen above, receives a `name` parameter, which corresponds again to the name of the SystemC-AMS TDF cluster from which the address needs to be obtained.

gpio2vci_address.c: The implementation of the `get_address` function is done here. This file is generated dynamically when the code of the software design level is generated. An example of this file is given in Listing 5.15, which was generated specifically for the two TDF clusters from Figures 5.2 and 5.3 of the model. In line 4, the `name` parameter is being compared to the name of the first TDF cluster `Cluster0`. If the names match, then the address of the GPIO2VCI component connected to that TDF cluster is returned in line 5 (Refer the assignment of segments of the mapping table from Listing 5.6 and the Net-List in Listing 5.9). If not, in line 6 the `name` is compared against the second TDF cluster, returning its address if the names match (line 7). If the `name` does not match to any of the TDF clusters from the model, then an error message will be printed to the TTY console of the model (line 9).

```
#ifndef GPIO2VCI_ADDRESS_H
#define GPIO2VCI_ADDRESS_H
#include <string.h>
#include <stdio.h>

int get_address(char name[]);

#endif //GPIO2VCI_ADDRESS_H
```

Listing 5.14: `gpio2vci_address.h` definition file from the `libsyscams` library.

```
#include "gpio2vci_address.h"

int get_address(char name[]) {
    if(strcmp(name, "Cluster0") == 0) {
        return 0xc0200000;
    } else if(strcmp(name, "Cluster1") == 0) {
        return 0xc1200000;
    } else {
        printf("ERROR getting address for cluster: \"%s\"\n", name);
        return -1;
    }
}
```

Listing 5.15: `gpio2vci_address.c` implementation file from the `libsyscams` library, dynamically generated.

libsyscams.config: This configuration file configures the `libsyscams` library. For this, also the file `config_noproc` was modified to include the `CONFIG_LIBSYSCAMS` variable.

Makefile: A Makefile was created to include the output objects from the two library files.

To generate the `gpio2vci_address.c` implementation file during the embedded software code generation, the following `.java` file was created.

Gpio2VciAddress.java: This file uses the information from the TDF clusters created in the SystemC-AMS Component Diagrams, in order to retrieve their addresses and names.

Figure 5.5 shows the C code that was written manually to the `readCluster1` state block. It shows how the `read_gpio2vci` function can be used to communicate with a specific cluster by passing only its name as a parameter. In this way the memory addresses of the clusters are abstracted to the final user.

Finally, in order to be able to compile the SystemC-AMS generated code, the SoCLib generated code, and the embedded software generated code all together using the `soclib-cc` commands, and to build the virtual prototype of the model, the following Makefile was modified.

Makefile.forsoclib: This Makefile copies all the necessary files that SoCLib needs in order to build the platform and runs the required compilation commands for this. The new files that are being generated as part of this work were added to this Makefile, so that the platform can be created and executed correctly.

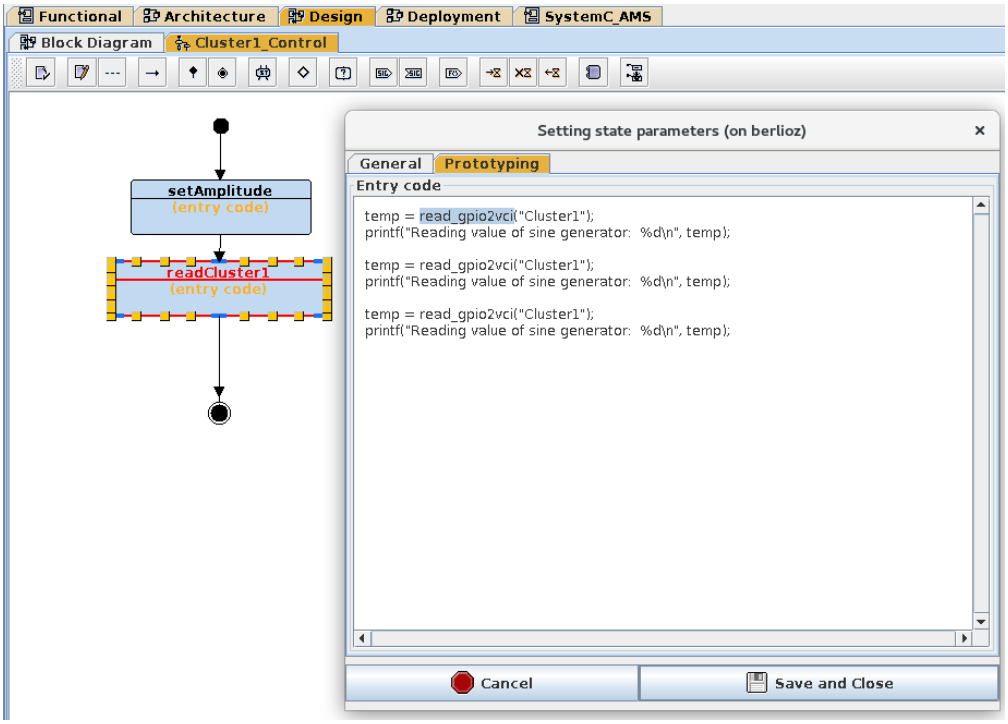


Figure 5.5: C code for the `readCluster1` state block.

```
 Cluster11.B1_1 @ 913206 ns: 0
 Cluster11.B1_1 @ 913207 ns: 0
@913207 ns: IDLE CMD_WRITE: 17
 Cluster11.B1_1 @ 913208 ns: 4.22773
READ-WRITE rspack
 Cluster11.B1_1 @ 913209 ns: 4.74285
 Cluster11.B1_1 @ 913210 ns: 5.25329
 Cluster11.B1_1 @ 913211 ns: 5.75854
 Cluster11.B1_1 @ 913212 ns: 6.25812
 Cluster11.B1_1 @ 913213 ns: 6.75151
 Cluster11.B1_1 @ 913214 ns: 7.23825
 Cluster11.B1_1 @ 913215 ns: 7.71784
 Cluster11.B1_1 @ 913216 ns: 8.18981
 Cluster11.B1_1 @ 913217 ns: 8.6537
 Cluster11.B1_1 @ 913218 ns: 9.10906
 Cluster11.B1_1 @ 913219 ns: 9.55542
 Cluster11.B1_1 @ 913220 ns: 9.99235
 Cluster11.B1_1 @ 913221 ns: 10.4194
 Cluster11.B1_1 @ 913222 ns: 10.8362
 Cluster11.B1_1 @ 913223 ns: 11.2423
 Cluster11.B1_1 @ 913224 ns: 11.6373
 Cluster11.B1_1 @ 913225 ns: 12.0208
 Cluster11.B1_1 @ 913226 ns: 12.3925
 Cluster11.B1_1 @ 913227 ns: 12.7519
 Cluster11.B1_1 @ 913228 ns: 13.0987
 Cluster11.B1_1 @ 913229 ns: 13.4326
 Cluster11.B1_1 @ 913230 ns: 13.7533
 Cluster11.B1_1 @ 913231 ns: 14.0604
```

Figure 5.6: Simulation output of console from the host machine

Once the virtual platform is created, it can be executed directly from TTool. In Figure 5.6 the output of the simulation from the console of the host machine is presented. It shows the print messages from the AMS Sink module (module **B1**), who is printing the values generated by the Sine Source (module **A1**) to the console of the local host machine. Note that the values are 0, but in time 913207 ns, a value of 17 is written to the GPIO2VCI component. After that, the Sine Source starts generating a sine wave of amplitude 17, which can be seen in the next lines of this output. In Figure 5.7 the output of the TTY

component from the model is shown. The first lines correspond to debugging messages from the CPU component. Note that in line 8, a message "Entering state + setAmplitude" is shown. Then the value 17 which was sent to the GPIO2VCI component in this state block is written. In the next line a message "Entering state + readCluster1" is printed. This means that the state block `readCluster1` will execute its code shown in Figure 5.5. In the next lines we see that three values from the Sine Source are read and printed in the TTY console component from the SoCLib modules.

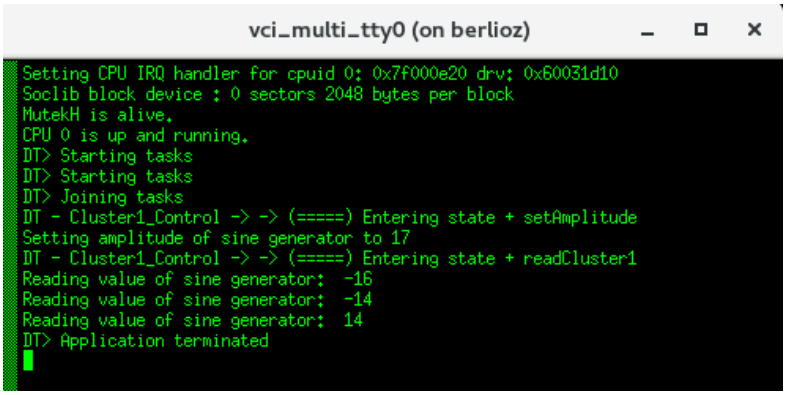


Figure 5.7: Simulation output of the terminal from the TTY component of the model.

5.3. Synchronization of SystemC-AMS and SoCLib modules in TTool

Since the SystemC-AMS simulation kernel runs a TDF simulation time (t_{TDF}) which is independent from the DE simulation time (t_{DE}) of the SystemC simulation kernel, synchronization of both simulation times is required. As explained in Chapter 4, time synchronization issues may occur and generate causality problems when dealing with multi-rate TDF blocks that are connected to DE blocks by means of TDF converter ports. As it was showed, these issues may happen only when there was an access to an input converter port that advanced the t_{DE} further than the t_{TDF} of the actual output converter port that is being accessed.

The synchronization between the DE and TDF MoCs should be validated before the SystemC-AMS code is generated. In other words, this task should be carried out during the design of the TDF model of the AMS components, in the SystemC-AMS Component Diagram.

The graphical interface was augmented to include a new panel, named Validation, inside the code generation window, as it is shown in Figure 5.8. By clicking the start button, the validation of the TDF model will be performed. This new panel was added in `JDialogSysCAMSExecutableCodeGeneration.java` under the `initComponents` method. This file is used to handle the functions of the Executable Code Generation dialog window for the SystemC-AMS Component Diagrams.

Some preliminary tasks should be performed before the actual synchronization between the DE and TDF MoCs can be done. Each of the TDF and DE ports from the TDF cluster should be connected to another port. This task should be validated as an initial step. This is done inside the `run` method in `JDialogSysCAMSExecutableCodeGeneration.java`. If

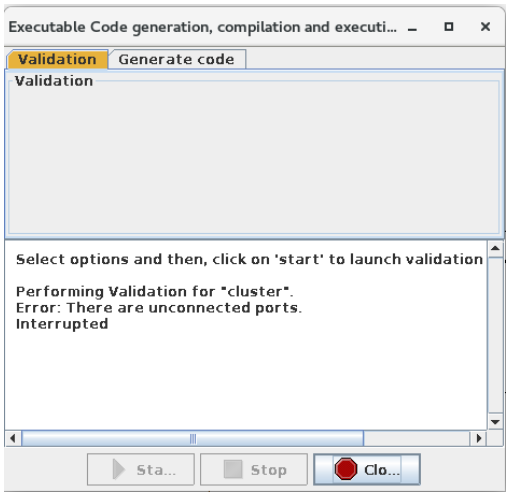


Figure 5.8: Validation panel and error for unconnected ports.

any port is not connected, an error message will be shown in the Validation panel of the code generation window, as it is shown in Figure 5.8

After validating that each port is connected to another port, the next step is to perform the TDF modules and ports timestep assignment and propagation. Figure 5.9 shows an example of how timestep propagates, as it is explained in [13, p. 11 f]. In this example all the port rates have been set to 1. The dotted blue arrows represent the timestep progpagations. Starting from the input port of module C, a Port-Timestep Tp = 10 ms (in black) has been assigned to it. The first propagation occurs to the Module-Timestep Tm of module C (number 1), which is set to 10 ms. Then, the input port Timestep from module C is propagated to the output port of module B (number 2), whose Tp is set to 10 ms. After this, the Tp of the output port of B propagates to the Module-Timestep Tm of module B (number 3), which is set to 10 ms. Then this Tm is propagated to the input port of B (number 4), whose Tp is set to 10 ms. Next, the timestep Tp of the input port of module B is propagated to the output port of module A (number 5), whose Tp is set to 10 ms. Finally the Tp of the output port of A propagates to the Tm of module (number 6).

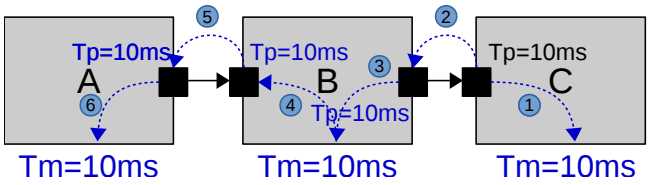


Figure 5.9: Timestep propagation (Adapted from [13, p. 11]).

As mentioned in [13], a consistency check should be made between the existing timesteps of the ports and modules, their rates, and the propagated timesteps. Two connected ports should have the same Port-Timesteps (Tp). And the Module-Timestep (Tm) should be consistent with the rate (R) and Tp of each of the ports that belong to that module. For

example, using module B from Figure 5.9, Equation 5.1 shows how timestep consistency should be checked for all its ports.

$$Tm_B = Tp_{B.in} \cdot R_{B.in} = Tp_{B.out} \cdot R_{B.out} \tag{5.1}$$

In the method `run` from `JDialogSysCAMSExecutableCodeGeneration.java`, the method `propagateTimestep` is called. This method initiates the timestep propagation of the TDF modules, and checks for consistency between the existing timesteps of the ports and modules, and the propagated timesteps. If there are any consistency errors, a message will be printed in the Validation panel from the code generation window, indicating the names of the blocks and ports where the consistency check failed, as it is shown in Figure 5.10

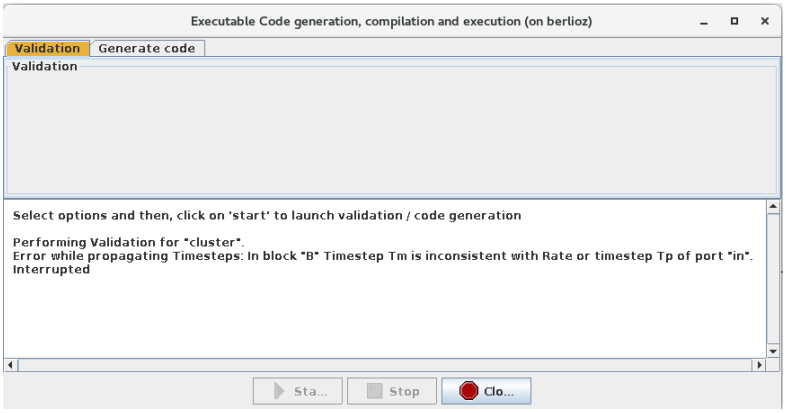


Figure 5.10: Timestep propagation error.

In order to validate the synchronization between the TDF and DE MoCs, a static schedule of the TDF cluster model needs to be computed first, as it was explained in Section4.3. Since the TDF MoC is based on the SDF modelling formalism, it can use the process of the SDF MoC to compute the static schedule. According to [25], the following elements are needed before the computation of the static schedule can be performed:

A topology matrix Γ needs to be built, where the number of columns of the matrix correspond to the number of TDF modules or nodes of an SDF graph; and the number of rows correspond to the number of signals connecting the TDF modules or arcs in an SDF graph. The entries of the matrix correspond to the rates of the ports of a module. If it is an output port, the entry is positive. If it is an input port, the entry is negative.

A buffer vector $b(i)$ represents the data available in the arcs of the graph in time i. The size of the buffer is the number of arcs of the graph. Note that $b(0)$ would represent the delays of the ports of the TDF modules, since at time $i = 0$ the delay samples will be available in the buffer.

The column vector q represents the number of executions of a node. In order to find the number of times that a node needs to be executed, the Matrix Equation 5.2 needs to be solved for q, and find the smallest positive integer solution.

$$\Gamma \cdot q = 0 \tag{5.2}$$

The sequential scheduling algorithm proposed in [25] is used to compute a static schedule of the TDF cluster. Listing 5.16 shows this algorithm.

```
1: procedure computeStaticSchedule
2:     Solve for the smallest positive integer vector q
3:     Form an ordered list L of the nodes (α) of the model.
4:     for each node α do
5:         if α is runnable then
6:             Schedule α
7:         end if
8:     end for
9:     if each node α has been scheduled q_α times then
10:        STOP
11:    else if no node could be scheduled then
12:        DEADLOCK
13:    else GO to 4
14:    end if
15: end procedure
```

Listing 5.16: Sequential Scheduling algorithm (Adapted from [25]).

In the `run` method in `JDialogSysCAMSExecutableCodeGeneration.java`, the method `buildTopologyMatrix` is called, which builds the topology matrix Γ and initiates the buffer vector $b(0)$. Then, the method `solveTopologyMatrix` is called, which solves the Matrix Equation 5.2 and provides the smallest positive integer solution for the equation. These steps correspond to line 2 of the algorithm. For line 3, an ordered list of nodes already exists for the modules of the TDF cluster. The method `computeSchedule` implements the steps from lines 4 - 14 of the algorithm. Note that in line 5, a node is considered runnable, if it has not run q times and it will not cause the buffer $b(i)$ to go negative. In line 12, a `DEADLOCK` implies that a schedule could not be found, because feedback loops exist in the graph and delays need to be added in order to find a schedule.

The `computeSchedule` method will detect if a deadlock is produced, and will compute a suggested delay (D_{sug}) that will solve the deadlock. The D_{sug} is calculated based on the last node α that tried to be scheduled and that made the buffer b_α to go negative. Equation 5.3 shows how the D_{sug} is calculated. It will use the current delay D_{cur} of the port associated to the buffer and subtract it from the value of buffer b_α. Remember that the buffer b_α was negative, so in this way a new delay will be calculated, which will make b_α to stop being negative. Once a delay is calculated, the scheduling algorithm from Listing 5.16 will run again to make sure that no other deadlocks exist.

$$D_{sug} = D_{cur} - b_\alpha \tag{5.3}$$

At this point, if the schedule cannot be computed, an error message will be shown in the Validation panel of the code generation window. The suggested delays for the TDF modules and ports that will make the model schedulable will be printed. The model shown in Figure 5.11 was created to show how a deadlock is detected and a delay to fix it is suggested. A loop exists between modules A and B. The port rates were set to 1 and no delays were introduced. The output of the Validation panel of the code generation window is shown in Figure 5.12.

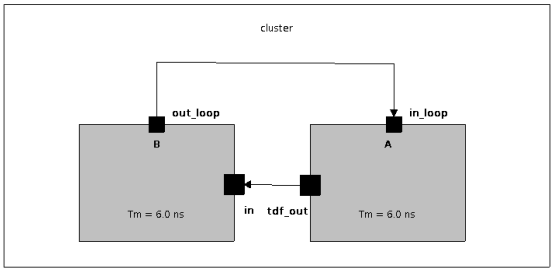


Figure 5.11: Unschedulable model due to missing delays in the loop.

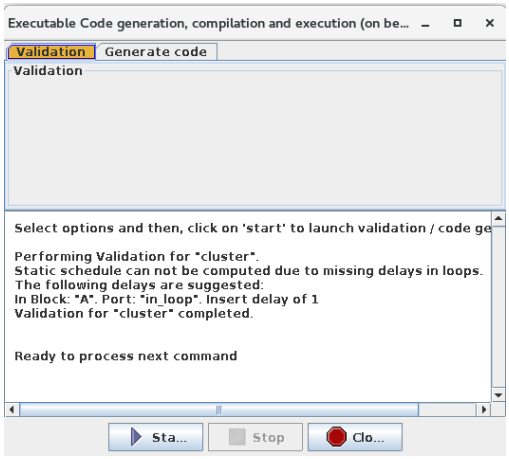


Figure 5.12: Suggested loop delays.

Now that a static schedule for a TDF cluster is computed, the synchronization between the DE and TDF MoCs can be performed. Note that in lines 4 and 5 from the scheduling algorithm of Listing 5.16, the algorithm is checking for each node if the node can be scheduled. This step will be used to call the procedure to detect time synchronization issues. A method `syncTDFBlockDEBlock` was created in `SysCAMSTBlockTDF.java`. This method is called from the `computeSchedule` method when a node has been found to be schedulable. The `syncTDFBlockDEBlock` method implements the proposed solution algorithm to detect time synchronization issues presented in Section 4.3, Listing 4.1. When a causality error is detected, a suggested delay is calculated. Then, the `computeSchedule` method is executed again using this new delay to verify if other causality problems exist. In this way, the `computeSchedule` method will run until all the time synchronization issues are resolved. If any causality problem occurs, an error message will be displayed in the Validation panel of the code generation window, showing the required delays for the TDF modules and ports that need to be added in order to solve the causality problems of the model.

The model presented in Chapter 4, was modeled in TTool to show the behavior of the implemented solution. Figure 5.13 shows the TDF cluster of the model in the SystemC-AMS Component Diagram. The first time it is validated, the model has no delays in any port. Figure 5.14 shows the output of the Validation panel of the code generation window. A message is printed suggesting that a delay should be inserted to solve the time synchronization issues. A delay of 1 in port `out_de` from module B is suggested. This is the same delay that was calculated in Section 4.2 for the same model. After the

suggested delay is inserted in the required port, the validation can be run again and then the SystemC-AMS code can be generated.

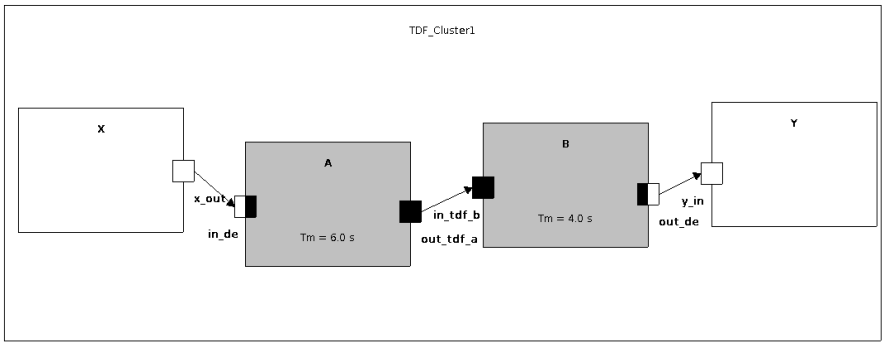


Figure 5.13: TDF cluster for time synchronization validation.

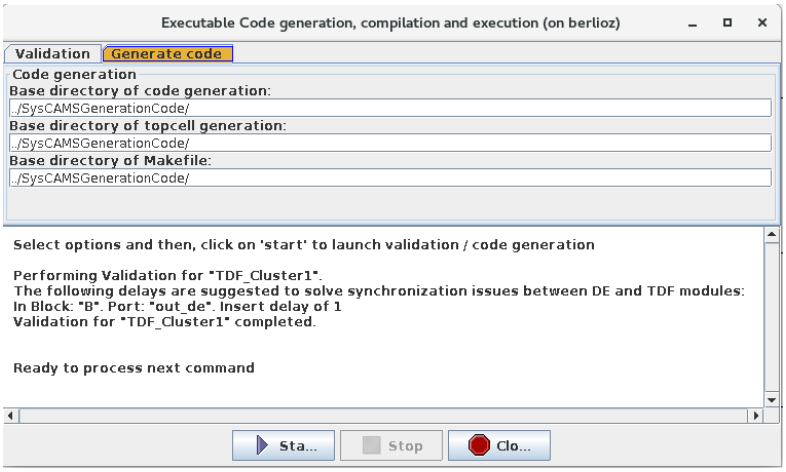


Figure 5.14: Time synchronization issues and suggested delays.

The integration of the SystemC-AMS modules into the SoCLib modules presented in this Chapter, allows the generation of virtual prototypes of heterogeneous embedded systems composed of digital and analog hardware, that can run embedded software using the MutekH OS. Moreover, it validates the correctness of the TDF models before the SystemC-AMS code is generated, giving suggestions to avoid problems in case that the model is unschedulable or if there is a chance for time synchronization issues to occur.

6. Case studies and comparison between TTool, SystemC-AMS and SystemC-MDVP

6.1. Case studies

6.1.1. Introduction

The following two case studies will be used to show the behavior of the implemented solution for the integration of SystemC-AMS and SoCLib modules into TTool. The first case study focuses on the generation of an heterogeneous virtual prototype of an embedded system that consists of digital and analog hardware components, and that can run software and an OS. The second case study focuses on the generation of an AMS virtual prototype in TTool starting from a TDF model, and comparing it with the results obtained from simulating the same model using the SystemC-AMS and SystemC-MDVP simulators.

6.1.2. Case study 1: Rover

The first case study is a rover system which is meant to assist rescuers to find victims in debris. The model consists of digital and analog components. A pure digital model built in [22] was modified.

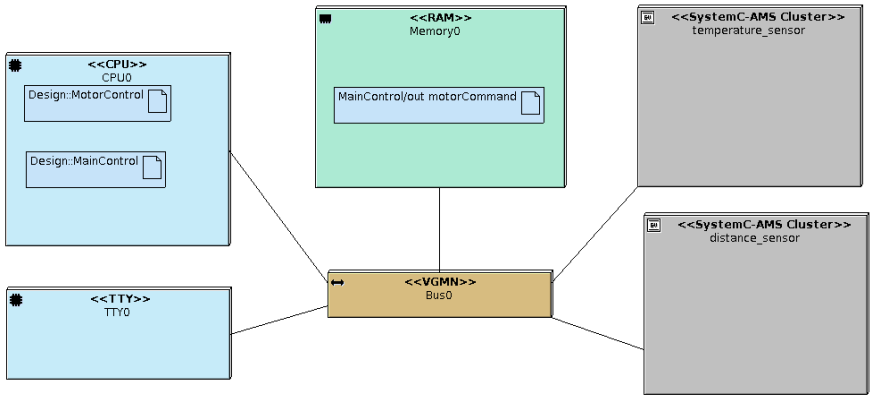


Figure 6.1: Deployment Diagram model of the rover

On the SoCLib part, a MIPS32 architecture CPU, a 1 MB RAM memory and a TTY terminal are modeled. On the analog part a distance sensor and a temperature sensor are modeled as two independent TDF clusters. Figure 6.1 shows the Deployment Diagram of the rover system where the SoCLib modules and TDF clusters are interconnected through a SoCLib interconnect component. In Figures 6.2 and 6.3, the temperature and distance sensor cluster models from the SystemC-AMS Component Diagram panels are shown.

The temperature sensor cluster shown in Figure 6.2, is composed of one TDF module modelling a temperature sensor unit. The behavior of this module, which is a simplification of the actual behavior of a temperature sensor, is described as SystemC-AMS code and shown in Listing 6.1. It depends on the value received on its input port `in`, which is

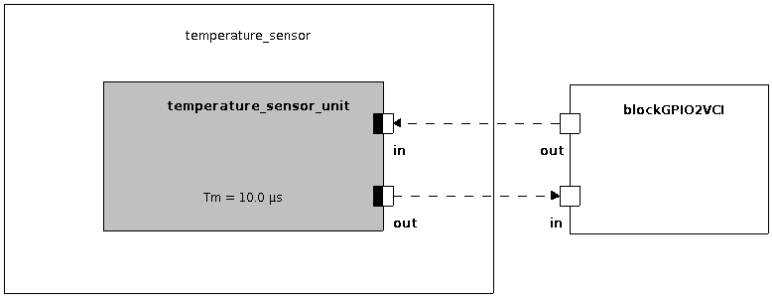


Figure 6.2: Temperature sensor model

connected to the digital components of the system via a GPIO2VCI component. A value of 0, means that the temperature sensor should be turned off, and no measure should be made (the module will print to the local host console a message stating that it is off). If a value different than 0 is received, then the temperature sensor will generate random integer values from 0 to 30, which represent the temperature measured from an object. This values are written to the output port `out` of the module which is connected to the GPIO2VCI component. Hence, the values will be available to be read by the digital components of the system. The temperature sensor will operate in timesteps of 10 μs.

```
1  void processing() {
2          if(in.read() != 0) {
3                  out.write(rand() % 30);
4          }
5          else {
6                  cout << "Temp sensor is off. @ " << this->get_time() << endl;
7          }
8  }
```

Listing 6.1: Temperature sensor behavior.

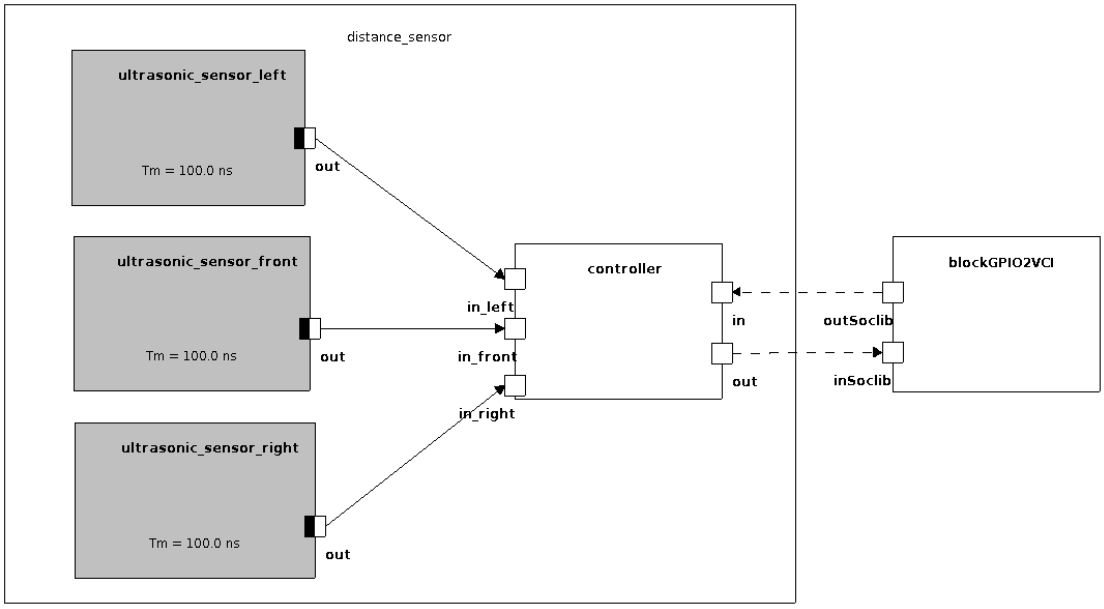


Figure 6.3: Distance sensor model

The distance sensor cluster shown in Figure 6.3, is composed of three TDF modules, which model an ultrasonic sensor unit each one; each of them connected to the same DE module, which models the distance sensor controller. This controller module reads values from each of the three ultrasonic sensors and writes a value to the GPIO2VCI component, depending on the value from its input port `in` that it receives from the digital components of the system via the GPIO2VCI component, as it shown in Listing 6.2. A value of 0 makes it read from the `ultrasonic_sensor_left`. A value of 1 makes it read from the `ultrasonic_sensor_front`. A value of 2 makes it read from the `ultrasonic_sensor_right`. Each ultrasonic sensor produces values in timesteps of 100 ns. The behavior of this cluster is a simplification of the behavior of a real distance sensor, since the interest of this work is mainly in the communication between the modules.

```
void read_sensor() {
        if(in.read() == 0) {
                out.write(in_left.read());
        }
        else if(in.read() == 1) {
                out.write(in_front.read());
        }
        else if(in.read() == 2) {
                out.write(in_right.read());
        }
}
```

Listing 6.2: Distance sensor controller behavior.

The ultrasonic sensor TDF modules simply generate an output random value from 0 to 12, which represents the distance to an object that is measured, as it is shown in Listing 6.3.

```
void processing() {
        out.write(rand() % 12);
}
```

Listing 6.3: Ultrasonic sensor behavior.

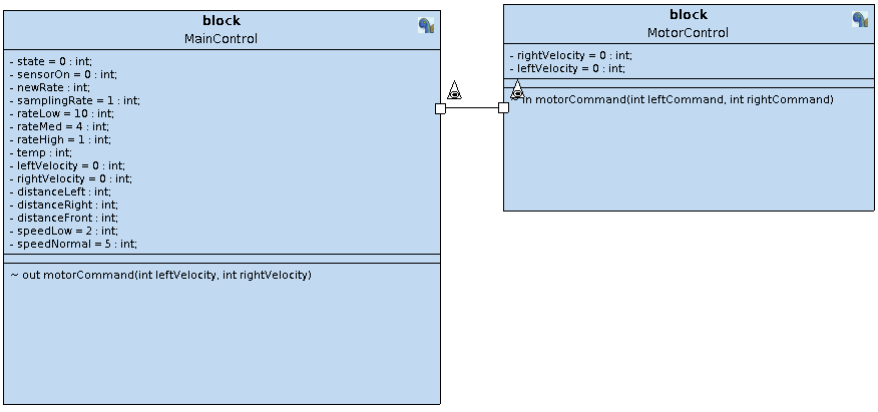


Figure 6.4: Block diagram for the rover

From the software side of the model, two blocks have been created to represent the `MotorControl` and the `MainControl` as shown in Figure 6.4. Both blocks communicate

with each other through a signal `motorCommand` which is sent by the `MainControl` to the `MotorControl` and contains two parameters for the right and left velocity to control the motor. The blocks also contain and initialize internal variables that the tasks of each block use.

The state machine of the `MotorControl` block, shown in Figure 6.5, has only one state `startMotor`. It receives the two velocity parameters from the `motorCommand` signal and waits for some random time between 10 and 20 clock cycles. This state machine only represents the actions that a motor controller would take to control the motor depending on the velocity parameters that it receives, but it actually does not have any functional behavior programmed, since the motor component is not modelled here.

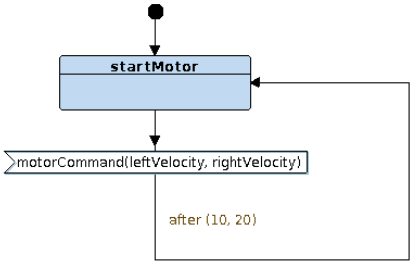


Figure 6.5: Motor control state machine.

The state machine of the `MainControl` block is shown in Figure 6.6. Below, a more detailed description from the states, specially from the ones that interact with the TDF clusters, is provided.

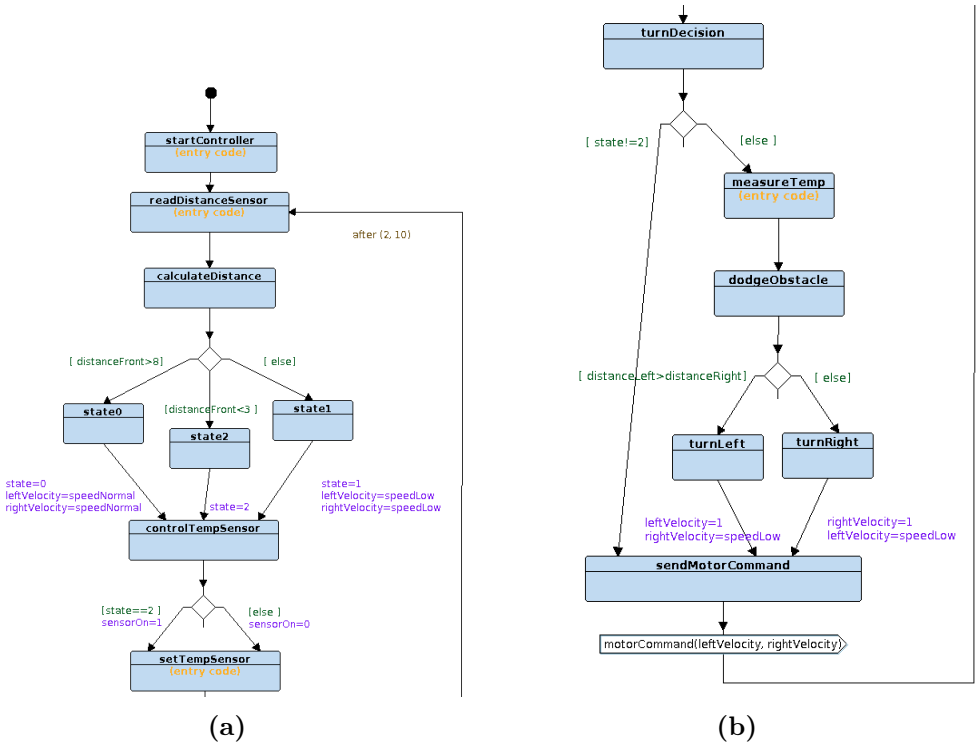


Figure 6.6: Main control state machine.

During the first state `startController`, the variable `sensorOn` which was initialized to 0 in the `MainControl` block shown in Figure 6.4, is written to the GPIO2VCI component

connected to the temperature sensor cluster. This is done by the C code written to the `startController` state shown in Figure 6.7. This will turn the temperature sensor unit module off. Figure 6.9 shows that the temperature sensor is already off, because no value is written yet to its input. Anyway, at 1581390 ns a value of 1 is written to the GPIO2VCI component, turning the temperature sensor unit off.

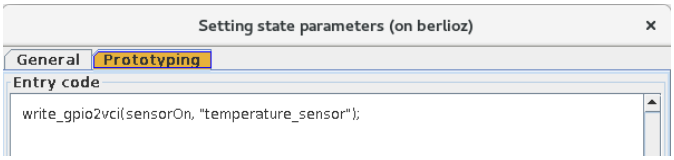


Figure 6.7: `startController` code.

In the next state `readDistanceSensor`, each ultrasonic sensor is selected by writing a different value to the distance sensor cluster, then the sensor output is read and the read value is printed to the TTY component of the model, as shown in the code from Figure 6.8. First, a value of 1 is written to read the front ultrasonic sensor. Then, a value of 0 is written to read the left ultrasonic sensor. Finally, the right ultrasonic sensor is read by writing a value of 2.

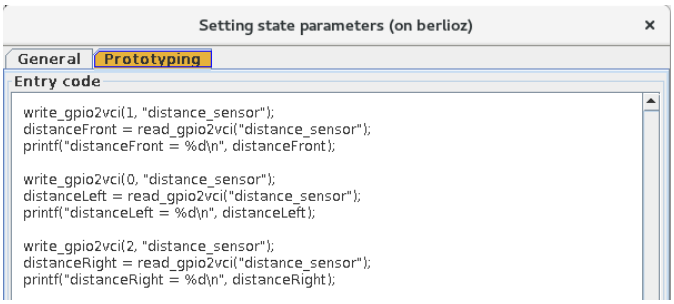


Figure 6.8: `readDistanceSensor` code.

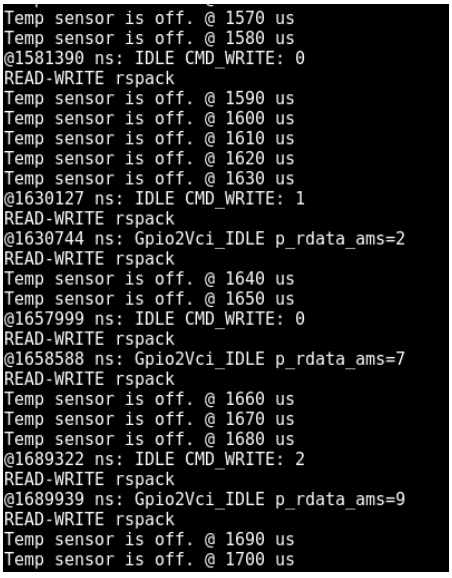


Figure 6.9: Rover simulation output from the host machine console.

In the local host console from Figure 6.9, it is shown that at time 1638127 ns a value of 1 is written and then a value of 2 is read. At time 1657999 ns a value of 0 is written and then a value of 7 is read. Finally at time 1689322 ns a value of 2 is written and a value of 9 is read.

The simulation output printed in the TTY component of the model also shows the three values that were read from the distance sensor. This can be seen in the lines inside the red square of Figure 6.10, where after entering the `startController` and `readDistanceSensor` states, a value of 2 is read from the front ultrasonic sensor; a value of 7 is read from the left ultrasonic sensor; and a value of 9 is read from the right ultrasonic sensor.

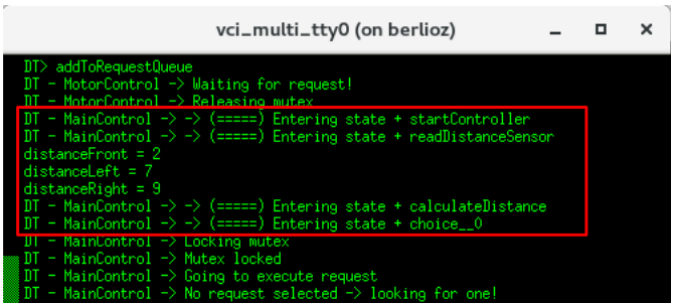


Figure 6.10: Rover simulation output from the TTY component console - `startController` and `readDistanceSensor` states.

After this, the next state `calculateDistance` simulates how the velocity of the rover is calculated based on the front distance that was read as shown in the state diagram of Figure 6.6a. If the distance was large (greater than 8), the state condition will lead to `state0` and a normal speed would be set (a value of 5). If the distance was between 3 and 8, it would go to `state1` and a low speed (a value of 2) would be set. In this case, since the front distance was 2, the `calculateDistance` state condition will lead to `state2`, since the distance is less than 3. Here the `state` variable is set to 2. Then it goes to the `controlTempSensor` state. Here depending on the `state` variable value, the `sensorOn` variable will be set. Since `state` = 2, `sensorOn` will be set to 1. Then the state machine goes to the `setTempSensor` state.

In the `setTempSensor` state, the temperature sensor unit will be turned on or off. This is done by writing to the temperature sensor cluster the value of the `sensorOn` variable as shown in the code from Figure 6.11. In the output from the local host machine, shown in Figure 6.12, at time 3196116 ns a value of 1 is written to the GPIO2VCI component. At this point the temperature sensor unit is turned on. We can see that the "Temp sensor is off" message in the local host machine console is not printed anymore.

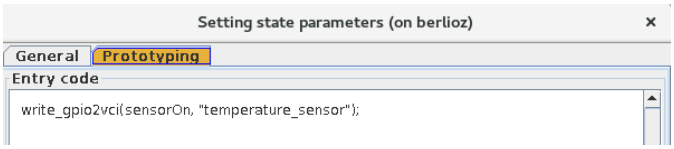


Figure 6.11: `setTempSensor` code.

In the following states shown in the state diagram from Figure 6.6b, the rover will measure the temperature and turn if it is close to an obstacle. Depending on the `state` variable,

```
Temp sensor is off. @ 3190 us
@3196116 ns: IDLE CMD_WRITE: 1
READ-WRITE rspack
@3935019 ns: Gpio2Vci_IDLE p_rdata_ams=21
READ-WRITE rspack
```

Figure 6.12: Rover simulation output from the host machine console.

the `turnDecision` state will decide if the rover needs to turn or not. If the distance is greater or equal than 3, then the `state` variable will be 0 or 1, and the rover will not measure tempreature or turn. In this case the distance is 2 and the `state` variable is 2, so it goes to the next state `measureTemp`.

In the `measureTemp` state, the temperature sensor cluster is read and the temperature is printed to the TTY component of the model, as shown in the code from Figure 6.13. In the local host console from Figure 6.12, at time 3935019 ns a value of 21 is read. Then, the output printed in the TTY component is showing that the temperature value that was read is 21, as shown in the lower red rectangle from Figure 6.14.

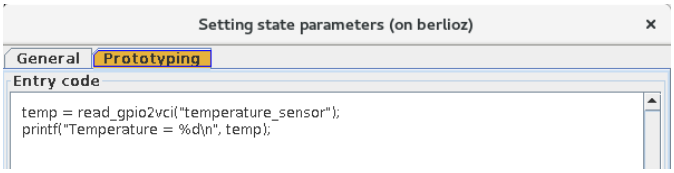


Figure 6.13: `measureTemp` code.

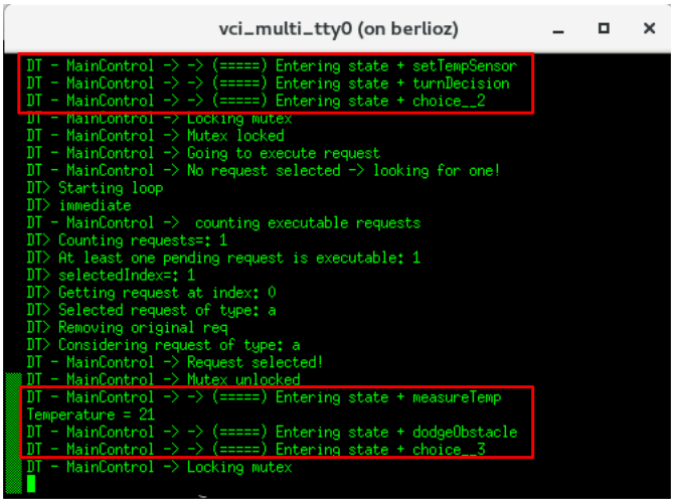


Figure 6.14: Rover simulation output from the TTY component console - `setTempSensor` and `measureTemp` state.

The `dodgeObstacle` state will calculate if it needs to turn left or right, based on the distance measured from the left and right ultrasonic sensors. To do this, it will set the velocity of the left or right motors accordingly. After turning left or right, the `motorCommand` signal will be sent to the motor control state machine, so that it can adjust the velocity of the motors of the rover. Finally, the main control state machine will loop again to the `readDistanceSensor` state to start a new cycle.

The generation of an heterogeneous virtual prototype that includes analog and digital hardware components has been demonstrated with this case study. The generated plat-

form consists of a digital SoC based on SoCLib components connected to analog hardware components, which are modeled using SystemC-AMS code. The virtual prototype is capable of running software and the MutekH OS. By calling software functions, the CPU of the platform is able to write or read values from the analog components.

6.1.3. Case study 2: Vibration sensor

The second case study is based on a vibration sensor model taken from the TDF model examples provided in the SystemC-MDVP simulator. The model of the vibration sensor is shown in Figure 6.15. It consists of six TDF modules and one DE module as described below.

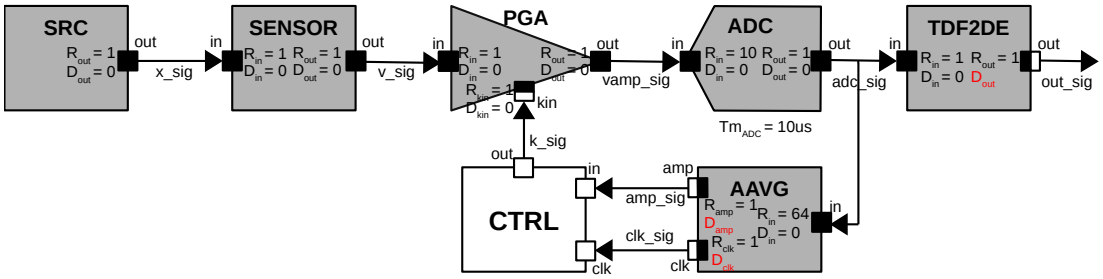


Figure 6.15: Vibration sensor model (Adapted from [7]).

The **SRC** module represents the vibration source, which is modeled as a generator of harmonic sinusoidal wavelets which represent a displacement signal (**x_sig**) caused by the vibration.

The **SENSOR** module represents a vibration sensor. It takes as an input the displacement signal (**x_sig**) and gives as an output a voltage signal (**v_sig**) which is proportional to the vibration velocity.

The **PGA** module represents a programmable gain amplifier, that amplifies the voltage input signal (**v_sig**) by a factor of 2^k, where k is the input value from signal **k_sig**. This signal is controlled by the gain controller DE module **CTRL**. As an output, it gives an amplified voltage signal **vamp_sig**.

The **ADC** module represents an analog to digital converter with `Nbits` resolution of 5. The ADC has a rate of 10 in its input port `in`. Hence, it takes 10 samples from the amplified voltage signal **vamp_sig** and produces a digitized integer value of N-bits (**adc_sig**) where the most significant bit corresponds to the sign. The Module-Timestep is assigned to this module as 10μs.

The **TDF2DE** module is a converter module from the TDF signal **adc_sig** to a DE signal **out_sig**. Note that the delay D_{out}, written in red, of its output converter port `out` has not been set yet.

The **AAVG** module represents an absolute amplitude averager. It calculates and outputs to the **amp_sig** the absolute average amplitudes of the received samples from the **adc_sig**. As it can be seen, its input port `in` has a rate of 64, meaning that it will receive 64 samples to calculate the absolute average amplitude. This module also generates a

clock signal **clk_sig** at its output port `clk`, which has a rate of 2, meaning that a clock edge will be generated twice per activation of the module. Note that the delays D_{clk} and D_{amp}, written in red, of its output converter ports have not been set yet.

The **CTRL** DE module represents the gain controller. This controller is modelled based on the state machine diagram shown in Figure 6.16. It will control the output signal **k_sig** based on the calculated absolute average amplitude given by **amp_sig**, and two given thresholds *low_threshold* and *high_threshold*, whose values are calculated as shown in Equations 6.1 and 6.2.

$$low\_threshold = 0.2 \cdot 2^{\mathrm{Nbits}-1} \tag{6.1}$$

$$low\_threshold = 0.8 \cdot 2^{\mathrm{Nbits}-1} \tag{6.2}$$

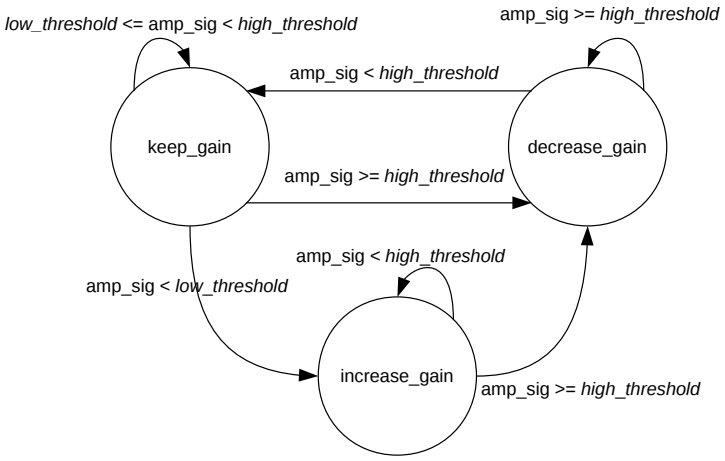


Figure 6.16: CTRL module state machine.

The vibration sensor was modelled in TTool, as shown in Figure 6.17. For a first validation, the three output converter port delays (in red) from Figure 6.15, were set to 0. In Figure 6.18 the output of the Validation panel of the code generation window is shown. Here, the time synchronization issues for this model were found and three suggested delays to solve the causality problems are shown.

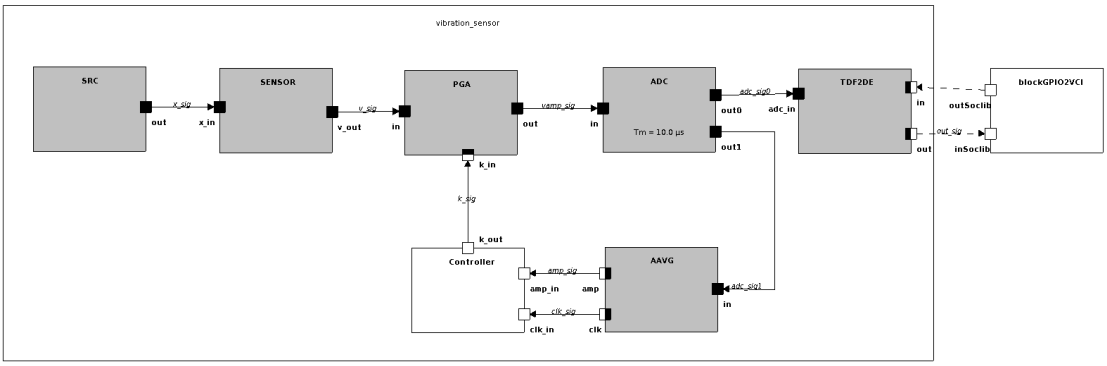


Figure 6.17: Vibration sensor model in TTool.

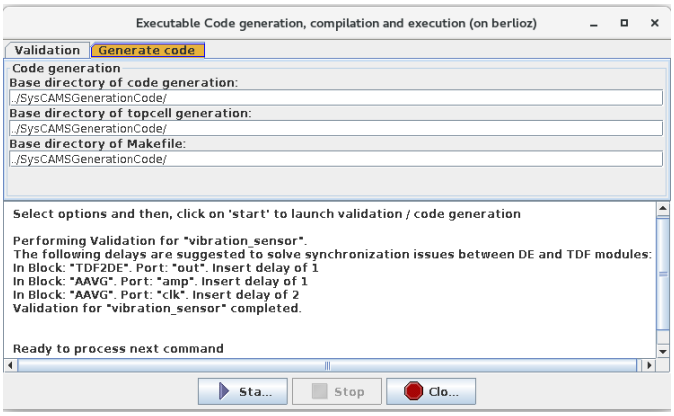


Figure 6.18: Suggested delays for the vibration sensor in TTool.

The vibration sensor model is already included in the SystemC-MDVP simulator, as part of the model examples. The model was simulated without providing any delays for its output converter ports. Figure 6.19 shows the output of the simulator. It suggests the same three delays as the ones suggested by TTool to solve the causality problems.

```
Incomplete schedule determined for the TDF cluster:
 · t = 0 s · Read k_sig
 · t = 0 s · SRC
 · t = 0 s · SENSOR
 · t = 0 s · SRC
 · t = 0 s · PGA
 · t = 0 s · SENSOR
 · t = 0 s · SRC

Delay changes suggested for solving the synchronization problems found in TDF converter ports during the elaboration phase:

     |-- port name       = TDF2DE.out
     |-- current delay   = 0
     |-- suggested delay = 1

     |-- port name       = AAVG.clk
     |-- current delay   = 0
     |-- suggested delay = 2

     |-- port name       = AAVG.out
     |-- current delay   = 0
     |-- suggested delay = 1
```

Figure 6.19: Suggested delays for the vibration sensor in SystemC-MDVP.

Finally, the SystemC-AMS model taken from a Master's course in context of the H-Inception project was modified to include the same parameters as the ones used in TTool and SystemC-MDVP—i.e. the same port rates and ADC resolution. The simulation was executed without assigning any delays to the output converter ports. As it is shown in Figure 6.20, synchronization issues are detected by the simulator each time the simulation is run, and delays referring to time units are suggested to solve the causality problems. For the first time the simulation was run, a delay of 9 μs in port `tdf2de.out` is suggested as it is shown in Figure 6.20b. This delay corresponds to a delay of 1 since the propagated timestep of this port is 10 μs. After setting this delay, the simulation was run again. This time another synchronization problem was found, and a delay of 639 μs is suggested to the `aavg.clk` port, as Figure 6.20b shows. Since the timestep of this port is of 320 μs, a delay of 2 is needed. Finally, after setting this new delay, the simulation was run for the third time. This time, another causality problem was detected, and a delay of 639 μs is suggested to the port `aavg.amp`. The timestep of this port is of 640 μs, so a delay of 1 is required.

All the suggested delays by SystemC-AMS simulator are the same as the ones suggested by TTool and the SystemC-MDVP simulator, as shown in Table 6.1. The big difference is that in TTool, the causality problems can be found at the design level before any code is

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scam
s/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 9 us
sca-next-time:   10 us insert da delay of at least: 9 us in: tdf2de.out
```

(a)

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scams
/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 639 us
sca-next-time:   320 us insert da delay of at least: 639 us in: aavg.clk
```

(b)

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scams
/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 639 us
sca-next-time:   640 us insert da delay of at least: 639 us in: aavg.amp
```

(c)

Figure 6.20: Suggested delays for the vibration sensor in SystemC-AMS.

generated. In SystemC-MDVP, the synchronization issues are found in the pre-simulation phase. That means that the SystemC-MDVP model needs to be executed only once to find any synchronization problems. In SystemC-AMS, these issues are found during the simulation phase, meaning that the simulation needs to be executed once per causality problem that is found. In this case, it needed to be executed three times.

Port	TTool	SystemC-MDVP	SystemC-AMS
TDF2DE.out	1	1	1
AAVG.amp	1	1	1
AAVG.clk	2	2	2

Table 6.1: Comparison of the suggested delays between TTool, SystemC-MDVP and SystemC-AMS for the vibration sensor model.

Once all the delays are set, the validation in TTool is run again, and no more issues were found. After this, the code generation can be executed. A simple SoC model was created as shown in Figure 6.21, where one SystemC-AMS Cluster block representing the vibration sensor was created. The code was generated with the tracing functionality enabled, so that it can generate a trace file of the vibration sensor signals. Finally the code was executed to run the SystemC simulation.

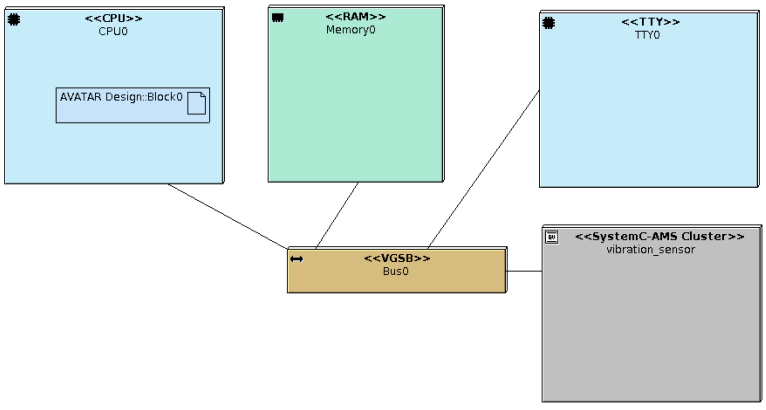


Figure 6.21: Deployment Diagram model including the vibration sensor TDF cluster.

In SystemC-AMS and SystemC-MDVP, all the delays were set as well, and the simulation was run, creating a trace file of the model's signals too, so that the three results could be compared. Figure 6.22 shows the analog waveforms in blue, resulting from

the simulation of the SystemC-AMS model. Figure 6.23 shows the analog waveforms in green, resulting from the simulation of the SystemC-MDVP model. Figure 6.24 shows the analog waveforms in red, resulting from the simulation of the SystemC model created from TTool. It can be seen that the three outputs match, specially for the fourth signal, which corresponds to the digitized output from the **ADC** component. The first waveform corresponds to the signal **x_sig** which carries the output of the harmonic sinusoidal wavelets generator **SRC**, simulating a vibration source. The second waveform is from the signal **v_sig**, which is the voltage output from the vibration sensor module **SENSOR**. The third waveform corresponds to the **v_amp** signal, which is the signal being amplified by the **PGA** module. The fourth signal **adc_sig** is the digitized output from the **ADC** module. The fifth signal **amp_sig** corresponds to the output of the absolute amplitude averager **AAVG** module which is connected to the DE controller **CTRL**. This controller gives the sixth signal **k_sig** which carries the factor that will be used by the **PGA** module to amplify the voltage signal **v_sig**. The last signal is the **clk_sig** used as clock signal for the controller **CTRL** module.

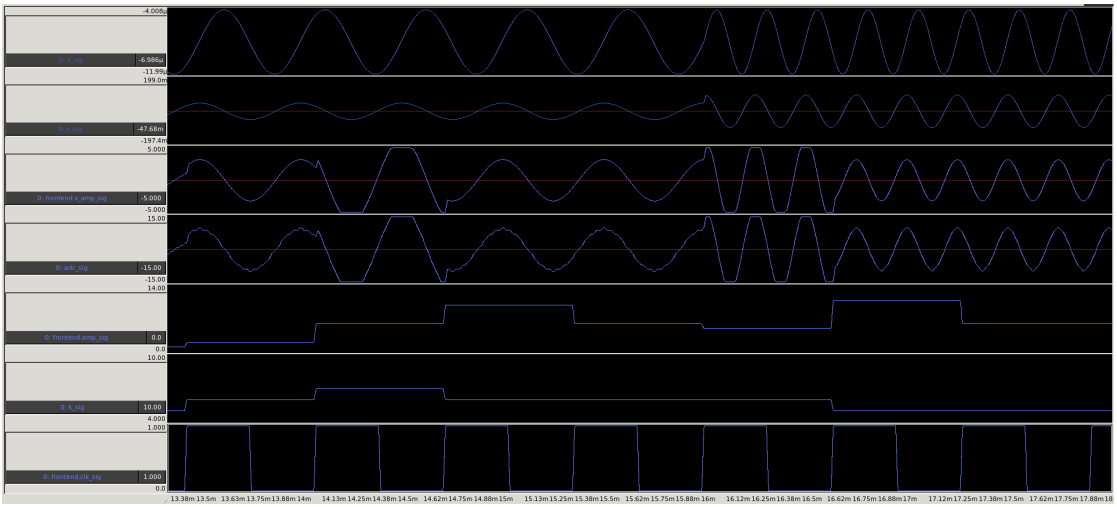


Figure 6.22: Vibration sensor trace signal generated from SystemC-AMS' simulation.

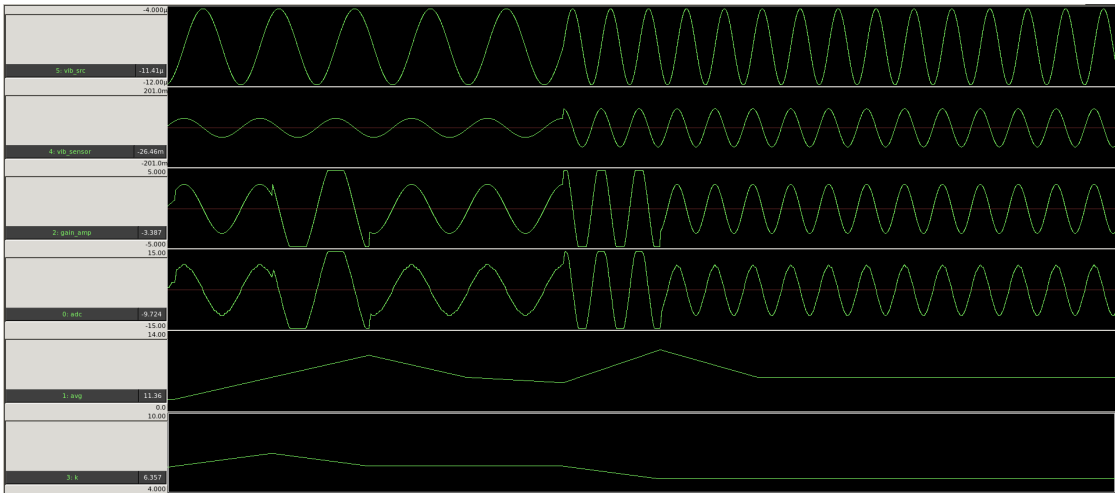


Figure 6.23: Vibration sensor trace signal generated from SystemC-MDVP's simulation.

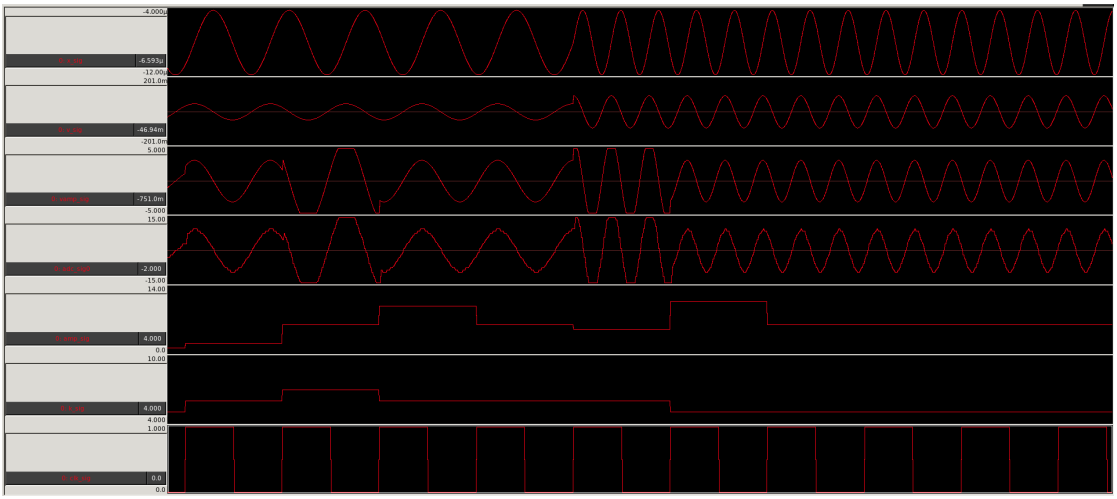


Figure 6.24: Vibration sensor trace signal generated from TTool's simulation.

Note that the **amp_sig** and **k_sig** signals from the SystemC-MDVP simulation look different, but this is due to the generated trace file that didn't created values when they were the repeated values. So only the value changes are shown, but still they correspond to the outputs from the other traces.

This second case study demonstrates that the solution implemented in TTool to detect time synchronization issues gives the same results as the ones suggested by the SystemC-AMS and the SystemC-MDVP simulators. Moreover, the time synchronization issues detection is performed at the design level, before the virtual prototype or the software code are generated. The case study also shows that the generated SystemC-AMS platform from TTool throws the same results as the models built directly in SystemC-AMS or in SystemC-MDVP.

6.2. Comparison between TTool, SystemC-AMS and SystemC-MDVP simulators

6.2.1. Model 1

The following model shown in Figure 6.25, will be used to compare the behavior of the three simulators using a model containing feedback and multiple DE components connected to a TDF module. This model was obtained from the SystemC-MDVP example models.

Note that only the Module-Timestep for module **A** is given. Regarding the port parameters, all the delays are set to 0. The Rate of the TDF ports and converter ports are shown below:

- A.out = 2
- B.in_tdf1 = 3
- B.in_tdf2 = 2
- B.out_tdf = 3
- B.in_de1 = 1

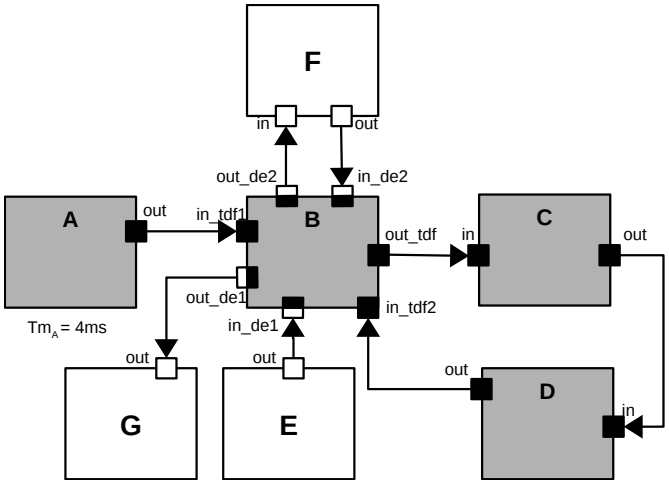


Figure 6.25: Model with feedback and multiple DE components connected to a TDF module.

- B.in_de2 = 3
- B.out_de1 = 6
- B.out_de2 = 1
- C.in = 2
- C.out = 4
- D.in = 3
- D.out = 1

The model was built as a TDF cluster in TTool, as it shown in Figure 6.26. After validating it, TTool found that the model could not be scheduled due to delays missing in the feedback loop, and also time synchronization issues occur within the converter ports of module **B**. This can be seen in the output from the Validation panel of the code generation window in Figure 6.27, where the necessary delays to solve these problems are suggested.

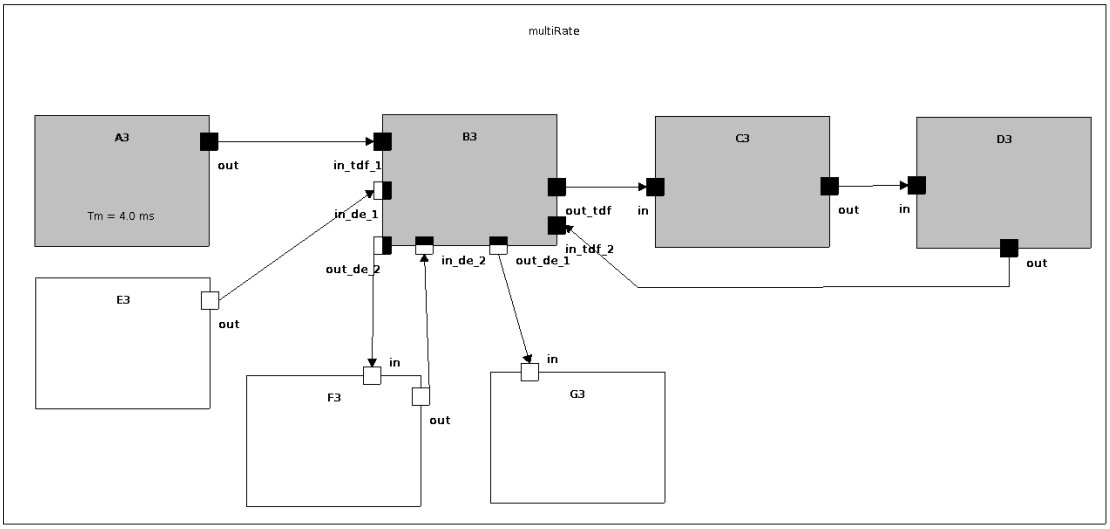


Figure 6.26: Model built in TTool.

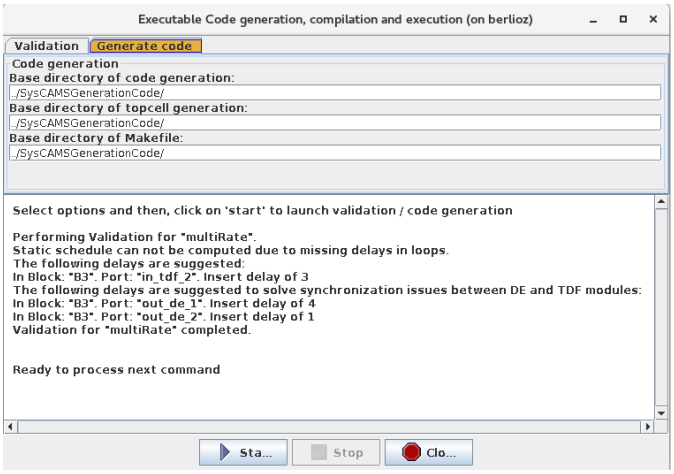


Figure 6.27: Suggested delays from TTool's model.

The same model was simulated in SystemC-MDVP, without including any delays. The results from the simulation are shown in Figure 6.28, where missing delays in the feedback loop and delays to solve causality problems are suggested.

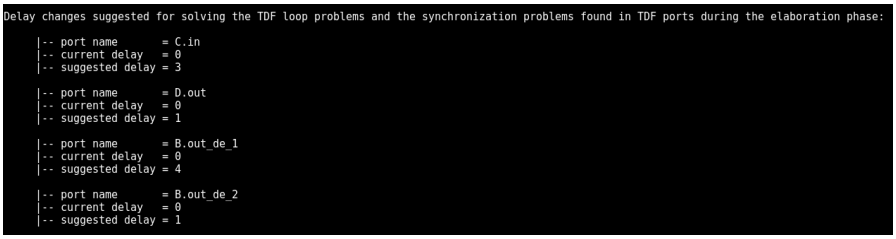


Figure 6.28: Suggested delays from SystemC-MDVP model's simulation.

Note that the suggested delays are different in both simulation outputs, as shown in Table 6.2. According to TTool's validation output from Figure 6.27, the suggested delays to B.out_de_1 = 4 and B.out_de_2 = 1, are needed to solve time synchronization issues. The SystemC-MDVP's simulation output from Figure 6.28 shows that these delays match. So the difference is between the delays that are suggested to make the model schedulable due to missing delays in the feedback loop. TTool suggests to insert a delay of 3 in the B.in_tdf_2 port, while the SystemC-MDVP simulator suggests to insert one delay of 3 in the C.in port and one delay of 1 in the D.out port.

Port	TTool	SystemC-MDVP
B.out_de_1	4	4
B.out_de_2	1	1
B.in_tdf_2	3	–
C.in port	–	3
D.out	–	1

Table 6.2: Comparison of the suggested delays between TTool and SystemC-MDVP.

The difference on the suggested delays lies in the static schedule that each tool produces. In the case of TTool, the static schedule produced for this model is

```
A-A-B-A-C-D-B-C-D-C-D-D
```

While in SystemC-MDVP the static schedule produced is

```
C-D-A-A-B-A-C-D-C-D-B-D
```

The static schedule from TTool shows that module **A** will be executed 2 times, and then module **B** will try to be executed. Since module **B** requires 2 samples in its port B.in_tdf2 to be executed, a delay of 2 is needed. The next time that module **B** needs to be executed, module **D** has only delivered 1 sample to **B**, so module **B** needs another delay of 1, in total it needs a delay of 3 in its input.

Now looking at the static schedule from SystemC-MDVP, module **C** needs to be executed first. Hence, it needs a delay of 3 in its input port C.in. Module **D** executes and delivers 1 sample to module **B**. But again, when module **B** needs to execute, it still needs one extra sample, so a delay in D.out of 1 is suggested.

The same model was created in SystemC-AMS without inserting any delays and simulated. Since there are missing delays in the feedback loop, the SystemC-AMS simulator output gives an error as shown in Figure 6.29, stating that the model cannot be scheduled, but no delays are suggested to fix the problem.

```
Info: SystemC-AMS:
       4 SystemC-AMS modules instantiated
       1 SystemC-AMS views created
       4 SystemC-AMS synchronization objects/solvers instantiated


Error: SystemC-AMS: System not schedulable - last schedulable element :
                mod_A3

    Current list:
                              mod_A3
                              mod_A3
                              mod_A3
```

Figure 6.29: Simulation of the model in SystemC-AMS, not schedulable.

To fix the unschedulable model problem, the suggested delays given by TTool were added to the SystemC-AMS model. The simulation was run again, and now the simulator found the time synchronization issues, as it is shown in the outputs from Figure 6.30, where the simulator was run twice. Figure 6.30a shows that the simulator suggests a delay of 4 ms in the output port B.out_de_1, whose timestep is of 1 ms, hence a delay of 4 is needed. Then in Figure 6.30b, a delay of 4 ms is suggested in the output port B.out_de_2, whose timestep is of 6 ms, hence a delay of 1 is needed. These are the same delays suggested by TTool and the SystemC-MDVP simulator. In this case, the static schedule produced by the SystemC-AMS simulator is

```
A-B-C-D-A-B-C-D-C-D-D-A
```

which is also different to both static schedules from TTool and SystemC-MDVP.

As a second test, the suggested delays given by the SystemC-MDVP simulator were added to SystemC-AMS model. The simulation was run, and the same time synchronization issues were found, giving the same delay suggestions to solve them. The only difference is that the static schedule produced by the SystemC-AMS simulator, was again different using these delays:

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scam
s/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 4 ms
sca-next-time:   1 ms insert da delay of at least: 4 ms in: mod_B3.out_de_1
```

(a)

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scam
s/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 4 ms
sca-next-time:   6 ms insert da delay of at least: 4 ms in: mod_B3.out_de_2
```

(b)

Figure 6.30: Suggested delays for time synchronization issues in SystemC-AMS.

```
A-C-D-A-B-C-D-A-C-D-B-D
```

Anyway, we can conclude that both delay suggestions given by TTool and the SystemC-MDVP simulator are equally valid, since they both solve the unschedulable model problem that exists when delays are missing in a feedback loop.

As a final test, the delays suggested by the SystemC-MDVP simulator were used in the TTool model. The model was validated correctly, without finding any other issues. Only, the static schedule from the TTool model changed to

```
C-D-A-A-B-A-C-D-C-D-B-D
```

Note that now this static schedule is the same as the one that the SystemC-MDVP simulator produces.

Finally, the delay suggestions given by the TTool validation were used now in the SystemC-MDVP model. In this case, the SystemC-MDVP simulator could not finish the simulation, since it seems it entered into a deadlock state.

From this model, it can be seen that when a model has feedback loops, the delay suggestions given by the SystemC-MDVP simulator and by TTool can be different between each other. They depend on the static schedule computed by each simulator. Anyway, both suggestions were found to be valid when they were used in the model created in SystemC-AMS. An interesting finding is that the delay suggestions given by TTool, could not be implemented in the SystemC-MDVP simulator.

6.2.2. Model 2

In this section, the model shown in Figure 6.31 is used to compare and explain the different delays suggested when the model is designed and validated in TTool. In the previous section it was shown that there are differences between the suggested delays for feedback loops given by TTool and the SystemC-MDVP simulator. These difference are due to the fact that the static schedules are computed in different ways in each tool.

As it was explained in Section 5.3, in order to build the static schedule, TTool uses the sequential scheduling algorithm of [25]. The algorithm uses an ordered list of the nodes to generate the schedule. TTool builds this ordered list based on the order in which the TDF blocks are created. Hence, this order will have an effect on the final static schedule and on the suggested delays to solve either the unschedulable model problems or the time synchronization issues.

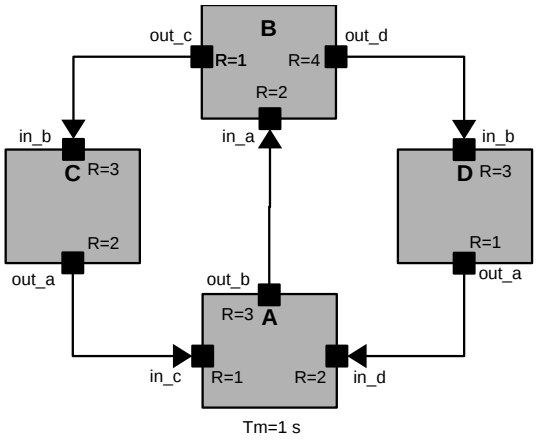


Figure 6.31: TDF model with two feedback loops (Adapted from [26]).

For example, the model was build first in TTool by creating the TDF blocks in this order: **A-B-C-D**. After validating the model, suggested delays of C.in_b = 3 and D.in_b = 12 were given, as shown in Table 6.3. In this case, the static schedule was built as follows:

```
C-D-D-A-B-D-D-A-B-B
```

A second model was built by creating the TDF blocks in this order: **D-C-B-A**. The model was validated and it suggested delays of A.in_d = 3 and A.in_c = 2 to solve the unschedulable problems. The computed static schedule for this test was:

```
A-B-D-A-B-D-B-D-C-D
```

A third model was created with the TDF blocks being inserted in the order: **A-D-C-B**. After validating the mode, a delay of B.in_a = 6 was suggested. For this test, the computed static schedule was:

```
B-D-B-D-B-D-C-A-D-A
```

Order of creation of the TDF blocks	Suggested delays
A-B-C-D	C.in_b = 3 D.in_b = 12
D-C-B-A	A.in_d = 3 A.in_c = 2
A-D-C-B	B.in_a = 6

Table 6.3: Suggested delays by TTool depending on the order of creation of the TDF blocks.

The three different scenarios produce valid schedules, and the suggested delays solve the unschedulable model problems. The model was created in SystemC-AMS, but as it was mentioned before, the simulator does not give any suggestion to fix the unschedulable

```
Info: SystemC-AMS:
        4 SystemC-AMS modules instantiated
        1 SystemC-AMS views created
        4 SystemC-AMS synchronization objects/solvers instantiated

Error: SystemC-AMS: System not schedulable - no element schedulable
```

Figure 6.32: Model simulated in SystemC-AMS with no delays.

model issue, as it is shown in Figure 6.32. But any of the delay suggestions made the model schedulable in SystemC-AMS.

Finally the model was built in the SystemC-MDVP simulator without introducing any delays. In this case, when the simulation was run, the simulator ended in an error as shown in Figure 6.33, and no delays could be suggested. And similar to the previous section, if the suggested delays from TTool are used, the simulator enters into a deadlock state.

```
        SystemC MDVP 1.0.0 --- Aug 30 2018 14:45:00
        Copyright (C) 2012-2015 by all Contributors,
        ALL RIGHTS RESERVED

Error: SystemC MDVP: Internal error printing an invalid schedule because the
CPN cpn_sca_simcontext.tdf_de_solver_0is computable: only a valid schedule ha
s been determined.

In file: detail/sca_cpn.cpp:608
```

Figure 6.33: Model simulated in SystemC-MDVP with no delays.

This model showed that the computed static schedule in TTool depends on the order that the TDF blocks are created. As it was explained with Model 1, the static schedule will influence the suggested delays given when the model is unschedulable because it contains feedback loops. Anyway, all the given suggestions will make the model schedulable. Comparing the results for this specific model, it was showed that TTool can give different delay suggestions to make the model schedulable. SystemC-AMS doesn't give any suggestions and the SystemC-MDVP simulator could not compute the model to present a valid suggestion.

6.2.3. Model 3

For this section, the model presented in Section 6.2.1 was extended to include some extra DE modules as shown in Figure 6.34. It will be used to show the effects on the suggested delays to solve time synchronization issues, when having different computed static schedules.

The Module-Timestep for module **A** is given as 4 ms. Regarding the port parameters, all the delays are set to 0. The Rate of the TDF ports and converter ports are shown below:

- A.in_de = 2
- A.out = 2
- B.in_tdf1 = 3
- B.in_tdf2 = 2
- B.out_tdf = 3
- B.in_de1 = 1
- B.in_de2 = 3

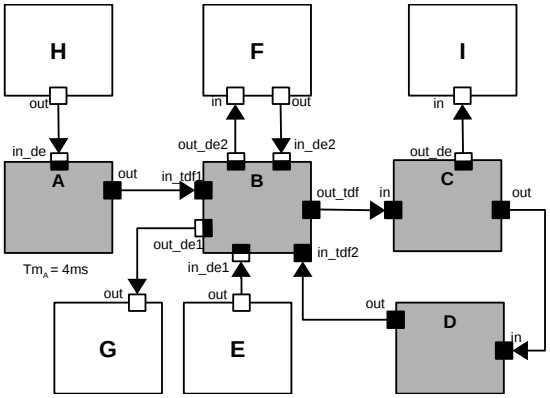


Figure 6.34: Extended Model 1 with extra DE modules.

- B.out_de1 = 6
- B.out_de2 = 1
- C.in = 2
- C.out = 4
- C.out_de = 1
- D.in = 3
- D.out = 1

The SystemC-MDVP model was modified to include the new extra DE modules. The model was simulated, and the output of the simulator shown in Figure 6.35, gives new suggestions on required delays to solve synchronization issues and to make the model schedulable.

```
Delay changes suggested for solving the TDF loop problems and the synchronization
 problems found in TDF ports during the elaboration phase:

     |-- port name       = C.in
     |-- current delay   = 0
     |-- suggested delay = 3

     |-- port name       = C.out_de
     |-- current delay   = 0
     |-- suggested delay = 1

     |-- port name       = D.out
     |-- current delay   = 0
     |-- suggested delay = 1

     |-- port name       = B.out_de_1
     |-- current delay   = 0
     |-- suggested delay = 6

     |-- port name       = B.out_de_2
     |-- current delay   = 0
     |-- suggested delay = 1
```

Figure 6.35: Suggested delays from the simulation in SystemC-MDVP.

As it was noted before, the delay suggestions in the ports C.in and D.out correspond to the necessary delays to make the model schedulable. These two delays are used in the models created for SystemC-AMS and TTool.

The same model was created in SystemC-AMS. In Figure 6.36, the outputs from the simulation suggesting delays to solve time synchronization issues are shown.

During the first run of the simulation, a delay of 2 ms in C.out_de is suggested as shown in Figure 6.36a. Since its timestep is of 4 ms, a delay of 1 is required. On the second run of the simulation, a delay of 6 ms is required on the port B.out_de1. Since its timestep

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scam
s/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 2 ms
sca-next-time:    4 ms insert da delay of at least: 2 ms in: mod_C3.out_de
```

(a)

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scam
s/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 6 ms
sca-next-time:    1 ms insert da delay of at least: 6 ms in: mod_B3.out_de_1
```

(b)

```
Error: SystemC-AMS: sca-de synchronization failed in: 0 ../../../../../source/src/scam
s/impl/core/sca_solver_base.cpp line: 529 current sca-time: 0 s current sc-time: 6 ms
sca-next-time:    6 ms insert da delay of at least: 6 ms in: mod_B3.out_de_2
```

(c)

Figure 6.36: Suggested delays from the simulation in SystemC-AMS.

is of 1 ms, a delay of 6 is needed. Finally, in the third run of the simulation, a delay of 6 ms is suggested for port B.out_de2. Its timestep is of 6 ms, hence a delay of 1 is required for this port. From these delays, it can be seen that they are the same delays that were suggested by the SystemC-MDVP simulator, as shown in Table 6.4.

Finally, the model was built in TTool as a TDF cluster. When the model was validated, the output from Figure 6.37, shows the suggested delays given by TTool in the Validation panel from the code generation window.

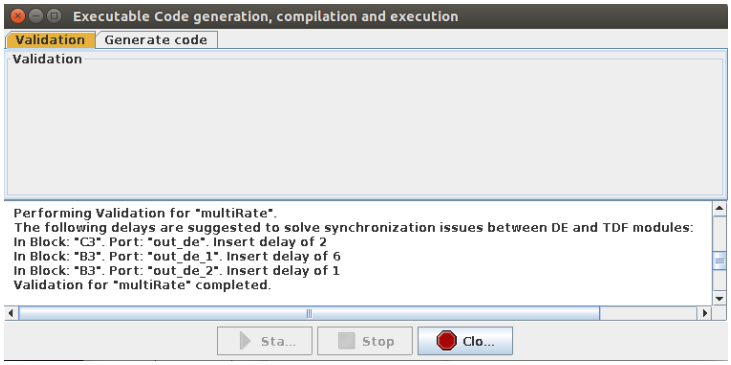


Figure 6.37: Suggested delays from the Validation panel in TTool.

The suggested delays for the ports B.out_de1 and B.out_de2 are the same as the ones suggested by SystemC-AMS and SystemC-MDVP simulators. But the delay suggested for port C.out_de is different. In TTool a delay of 2 is suggested, while in the other simulators a delay of 1 is suggested. The delay will for sure solve the time synchronization issues, even if it is not the optimal delay for this scenario. The reason for this difference, as mentioned before, is due to the static schedules that each simulator computes for the model.

Port	TTool	SystemC-MDVP	SystemC-AMS
C.out_de	2	1	1
B.out_de1	6	6	6
B.out_de2	1	1	1

Table 6.4: Comparison of the suggested delays to solve time synchronization issues between TTool, SystemC-AMS and SystemC-MDVP.

For SystemC-AMS, the static schedule is:

```
A-C-D-A-B-C-D-A-C-D-B-D
```

For SystemC-MDVP the computed static schedule is:

```
C-D-A-A-B-C-D-C-D-A-B-D
```

And for TTool, the static schedule was computed as:

```
C-A-D-A-B-A-C-D-C-D-B-D
```

The three of the schedules are different, and if they are compared to the schedules computed from Section 6.2.1, they are also different. From the model on Figure 6.34, observe that the module **A** accesses an input converter port, and the module **C** accesses an output converter port. These two modules are of our interest because they change their execution position in the three schedules; specially, they change their position relative to each other. Even if module **B** also accesses input and output converter ports, in the three schedules it keeps its position, and its position relative to the other two modules does not change.

In other words, when **B** is scheduled, in the three schedules it gets executed for the first time after 2 executions of **A** and 1 execution of **C** occur. Then, the second time **B** is executed, it appears after 1 more execution of **A** and 2 more executions of **C**, in the three schedules. For that reason it is said that B will not affect the causality even if the three schedules are different.

In the SystemC-AMS schedule, module **C** is always scheduled after one execution of module **A**. These means that module **A** will access its input converter port once, advancing the t_{DE}, and then module **C** will access its output converter port once, requiring a delay of 1 to solve the causality problems.

In the case of the SystemC-MDVP simulator, we see that module **C** is executed first, which causes no issues, because an access to an output converter port will not advance the t_{DE}, it will only advance its own t_{TDF}. Then the module **A** is accessed twice. This will advance t_{DE} twice. Later, module **C** will be executed. At this moment, module **C** only requires a delay of 1, since it was already executed before once and its t_{TDF} already advanced once. Finally, module **C** is executed again, advancing its t_{TDF}. Finally, when module **A** is executed for the last time, no more synchronization issues can occur.

In the case of TTool, we can observe that module **C** executes first, advancing its own t_{TDF} without advancing the t_{DE}. But then module **A** executes three times before the next execution of module **C**. This will make the t_{DE} to advance three times. So the next time module **C** is executed, a delay of 2 is needed to solve the time synchronization issues.

From this model, it can be concluded that TTool's delay suggestions to solve time synchronization issues also depend on the computed static schedule. Therefore in some specific scenarios, the delay suggestions given by TTool may not be the most efficient, if they are compared to the ones given by SystemC-AMS or SystemC-MDVP. Anyway, the given delay suggestions will solve the time synchronization issues between the DE and TDF MoCs.

7. Conclusion and perspectives

7.1. Conclusion

In this thesis, the integration of SystemC-AMS and SoCLib modules into TTool was implemented. This integration allows the generation of virtual prototypes of heterogeneous embedded systems, composed of SystemC-AMS models of analog hardware and a digital MPSoC platform based on SoCLib components. The generated virtual prototype can run embedded software and the MutekH OS.

The integration was developed in two stages. During the first stage, the SystemC-AMS and SoCLib components were integrated outside TTool. The GPIO2VCI SoCLib component was developed to cover the need of having a generic adaptor that could work as an interface between the SystemC-AMS and SoCLib components. Through this adaptor the SystemC-AMS components can be connected to the VCI interconnect component from SoCLib, allowing the integration of analog and digital components. At the end of this stage, a virtual prototype including analog components and a SoC platform, which could run software without using an OS, was generated.

In the second stage, the SystemC-AMS and SoCLib components were integrated into TTool. This stage was based on the previous work from [4], where SystemC-AMS TDF models could be created in TTool, using the SystemC-AMS Component diagrams, and SystemC-AMS code code be generated to simulate these models without any SoCLib digital component. To perform the integration, the generated SystemC-AMS code was modified so that the TDF clusters could be instantiated in the generated SoCLib topcell, and connected to the SoCLib interconnect component using the GPIO2VCI component. Since the generated virtual prototype should be able to run software and the MutekH OS to interact with the newly integrated AMS components, a MutekH library was created to give the software developers write and read functions to allow communication with the TDF clusters. At the end of this state, an heterogeneous virtual prototype composed of SystemC-AMS analog components and an MPSoC platform based on SoCLib components, able to run software and the MutekH OS, could be generated.

The SoCLib components code is generated in SystemC, which is based on the DE MoC. The analog components are modeled based on the TDF MoC. As it was explained, when there are interactions between the DE and TDF MoCs, time synchronization issues that generate causality problems may occur. One of the findings regarding the time synchronization issues, is that they can only occur when there is a read synchronization operation before a write synchronization operation. A solution for these issues has been proposed and implemented into TTool during the second stage of the integration. Also, when feedback loops exist in a TDF model and delays are not inserted within the loop, the model will become unschedulable. A solution to suggest which delays will make the model schedulable has also been implemented. Following TTool's philosophy, these solutions allow to validate the TDF models at the design level, before any code is generated.

It is important to note that the suggested delays to solve time synchronization issues and to make a model with feedback loops schedulable, are based on the computed static schedule. The static schedule computation implemented in TTool is based on the sequential scheduling algorithm proposed by [25]. The delays being suggested by the SystemC-AMS and the SystemC-MDVP simulators may be different to the ones proposed by TTool,

since their computed static schedules are different too. Anyway, the results show that the suggested delays by TTool are also valid, although they may not be the most efficient in every scenario. On the other side, it was shown that TTool gives suggestions to solve missing delays in feedback loops, which is not done by the SystemC-AMS simulator and in some specific models it leads to failures on the SystemC-MDVP simulator.

7.2. Perspectives

In this work, a library was created to provide read and write functions to the GPIO2VCI component, that can be used in the State Machine Diagrams from TTool in the software design level. To use these functions, manual C code needs to be written in the state blocks of the diagram. In the future, specific GPIO2VCI read and write state blocks can be implemented so that manual code is avoided, and correct-by-construction code is generated.

The GPIO2VCI component only allows single read and write communication with one TDF module per TDF cluster. If future models require to connect more than one module of a TDF cluster to the SoCLib components of an MPSoC platform, then the GPIO2VCI component will need to be extended to support multi-ports.

The AMS hardware components are considered to be targets inside the MPSoC platform. In the future, it may be required that these components act also as initiators, or they may need to support sending interruption requests to the CPUs.

The computed static schedule in TTool can be optimized, so that the suggested delays to solve time synchronization issues or to make a model with feedback loops schedulable are minimum. Another option is to compute the static schedule in a similar way as the SystemC-AMS simulator does, since the analog virtual platform code is being generated to be run by the SystemC-AMS simulator.

The graphical interface of TTool has been augmented to support the creation of ELN models as part of the work of [4]. The integration of the ELN MoC can be carried out in a future project.

The work developed during this thesis will be used in a master's project aimed to start on January 2019, which is part of the larger "EchOpen" project [27].

A. Appendix: Source codes

```
#ifndef GPIO2VCI_H
#define GPIO2VCI_H

#define sc_register sc_signal

#include <signal.h>
#include <stdlib.h>
#include <systemc.h>
#include "caba_base_module.h"
#include "vci_target.h"
#include "mapping_table.h"

namespace soclib {
namespace caba {

template <typename vci_param>
class Gpio2Vci
      : public soclib::caba::BaseModule {

    sc_register< typename vci_param::data_t > r_rdata_ams, r_wdata_ams;
    sc_register<int> r_fsm_state, r_buf_eop;

    enum fsm_state_e {
      TARGET_IDLE = 0,
      TARGET_WRITE,
      TARGET_READ,
    };

    const soclib::common::Segment   m_segment;

protected:
    SC_HAS_PROCESS(Gpio2Vci);

public:
    //Ports
    sc_in<bool>                             p_clk;
    sc_in<bool>                             p_resetn;
    soclib::caba::VciTarget<vci_param>      p_vci;
    sc_in< typename vci_param::data_t >     p_rdata_ams;
    sc_out< typename vci_param::data_t >    p_wdata_ams;

    Gpio2Vci( sc_module_name                insname,
        const soclib::common::IntTab        &index,
        const soclib::common::MappingTable  &mt );

    ~Gpio2Vci();

private:
    void transition();
    void genMoore();

};


}
}

#endif // GPIO2VCI_H
```

Listing A.1: GPIO2VCI definition code `gpio2vci.h`.

```
#include "../include/gpio2vci.h"
#include <iostream>

namespace soclib {
namespace caba {

#define tmpl(x) template<typename vci_param> x Gpio2Vci<vci_param>

tmpl(/**/)::Gpio2Vci( sc_module_name                    insname,
                      const soclib::common::IntTab      &index,
                      const soclib::common::MappingTable &mt )
        : soclib::caba::BaseModule(insname),
          m_segment(mt.getSegment(index)),
          p_clk("p_clk"),
          p_resetn("p_resetn"),
          p_vci("p_vci"),
          p_rdata_ams("p_rdata_ams"),
          p_wdata_ams("p_wdata_ams") {
    std::cout << "  - Building Gpio2Vci " << insname << std::endl;

    SC_METHOD(transition);
    sensitive << p_clk.pos();
    SC_METHOD (genMoore);
    sensitive << p_clk.neg();
}

tmpl(/**/)::~Gpio2Vci(){}

tmpl(void)::transition() {
    if(p_resetn == false) {
        r_fsm_state = TARGET_IDLE;
    }
    else {
        switch( r_fsm_state ) {
        case TARGET_IDLE:
            if( p_vci.cmdval.read() ) {
                r_buf_eop = p_vci.eop.read();
                if ( p_vci.cmd.read() == vci_param::CMD_WRITE ) {
                    r_wdata_ams = p_vci.wdata.read();
                    r_fsm_state = TARGET_WRITE;
                }
                else {  //VCI_CMD_READ
                    r_rdata_ams = p_rdata_ams.read();
                    r_fsm_state = TARGET_READ;
                }
            }
            break;

        case TARGET_WRITE:
        case TARGET_READ:
            if( p_vci.rspack.read() ) {
                r_fsm_state = TARGET_IDLE;
            }
            break;
        }
    }

}
tmpl(void)::genMoore() {
    switch (r_fsm_state) {
    case TARGET_IDLE:
        p_vci.rspNop();
        break;
    case TARGET_WRITE:
        p_vci.rspWrite( r_buf_eop.read() );
        p_wdata_ams.write(r_wdata_ams);
        break;
    case TARGET_READ:
        p_vci.rspRead( r_buf_eop.read(), r_rdata_ams );
        break;
    }
```

```
72      // We only accept commands in Idle state
73      p_vci.cmdack = (r_fsm_state == TARGET_IDLE);
74  }
75  }
76  }
```

Listing A.2: GPIO2VCI implementation code `gpio2vci.cpp`.

```
1     # -*- python -*-
2   Module('caba:gpio2vci',
3          classname = 'soclib::caba::Gpio2Vci',
4              tmpl_parameters = [
5                  parameter.Module('vci_param', default = 'caba:vci_param'),
6              ],
7          header_files = ['../source/include/gpio2vci.h',],
8          implementation_files = ['../source/src/gpio2vci.cpp',],
9          ports = [
10          Port('caba:clock_in', 'p_clk', auto = 'clock'),
11          Port('caba:bit_in', 'p_resetn', auto = 'resetn'),
12          Port('caba:vci_target', 'p_vci'),
13          ],
14         uses = [
15          Uses('caba:base_module'),
16          Uses('common:mapping_table'),
17          Uses('caba:vci_target'),
18          ],
19         instance_parameters = [
20          parameter.IntTab('ident'),
21          parameter.Module('mt', typename = 'common:mapping_table', auto = 'env:mapping_table'),
22          ],
23         )
```

Listing A.3: Code for the `gpio2vci.sd` Metadata file.

```
1   # -*- python -*-
2   todo = Platform('caba', 'top.cpp',
3                   uses = [
4       Uses('caba:vci_xcache_wrapper', iss_t = 'common:mips32el'),
5       Uses('caba:vci_ram'),
6       Uses('caba:vci_multi_tty'),
7       Uses('caba:vci_vgmn'),
8       Uses('common:elf_file_loader'),
9       Uses('caba:gpio2vci'),
10      ],
11      cell_size = 4,
12      plen_size = 6,
13      addr_size = 32,
14      rerror_size = 1,
15      clen_size = 1,
16      rflag_size = 1,
17      srcid_size = 8,
18      pktid_size = 1,
19      trdid_size = 1,
20      wrplen_size = 1
21  )
```

Listing A.4: Platform Description File `platform_desc` for the SoCLib SoC model.

```
# -*- python -*-
import os
syscams = os.getenv('SYSTEMC_AMS')
assert syscams, ValueError("Must set $SYSTEMC_AMS")
sysc = os.getenv('SYSTEMC')
assert sysc, ValueError("Must set $SYSTEMC")

config.systemc_ams = Config(
        base = config.systemc,
        cflags = ['-Iinclude',
              '-I'+sysc+'/include',
              '-I'+syscams+'/include'
          ],
        libs = ['-Wl,-rpath='+sysc+'/lib-linux64', '-L'+sysc+'/lib-linux64',
            '-Wl,-rpath='+syscams+'/lib-linux64', '-L'+syscams+'/lib-linux64',
        '-lsystemc-ams', '-lsystemc', '-lm'
        ],
        )
config.ams = Config(
    base = config.default,
    systemc = config.systemc_ams,
    repos = "./obj/soclib-cc",
    )
config.default = config.ams
```

Listing A.5: Code for the `soclib.conf` Configuration file.

```
# -*- python -*-

# SOCLIB environment definition
def mkname():
        try:
                import os
                import pwd
                return pwd.getpwuid(os.getuid())[0]
        except OSError:
                try:
                        import os
                        return os.environ["LOGNAME"]
                except KeyError:
                        return 'unknown'

config.toolchain_64 = Toolchain(
      parent        = config.toolchain,
      max_processes = 3,
      cflags        = config.toolchain.cflags+['-m64'],
      )

config.systemc_22_64 = Library(
      parent = config.systemc,
      dir    = "/users/outil/systemcams/systemc-2.3.1/",
      cflags = config.systemc.cflags,
      os     = "linux64",
      )

config.systemc_ams = Library(
        parent = config.systemc,
        dir    = "/users/outil/systemcams/systemc-ams-2.1",
        cflags = ['-Iinclude',
              '-I/users/outil/systemcams/systemc-ams-2.1/include'
          ],
        libs = ['-Wl,-rpath=/users/outil/systemcams/systemc-2.3.1/lib-linux64',
        '-L/users/outil/systemcams/systemc-2.3.1/lib-linux64',
        '-Wl,-rpath=/users/outil/systemcams/systemc-ams-2.1/lib-linux64',
        '-L/users/outil/systemcams/systemc-ams-2.1/lib-linux64',
        '-lsystemc-ams', '-lsystemc', '-lm'
        ],
        )

# Definition of a new build environments, which can be referenced with 'soclib-cc -t'

# SystemC 64bits environment
config.systemc_64 = BuildEnv(
      parent = config.build_env,
      repos = "/dsk/l1/misc/%s/tmp/soclib_repos_64"%mkname(),
      toolchain = config.toolchain_64,
      libraries = [config.systemc_22_64, config.systemc_ams],
      )

config.default = config.systemc_64
```

Listing A.6: Code for the `global.conf` Configuration file of SoCLib for TTool.

B. Appendix: Directory tree of source code and generated files

In this section, the directory tree of all source files modified for this project and all the generated files is shown. Listing B.1 shows the location of the automatically generated files from TTool. The SystemC-AMS generated files for the TDF clusters are stored under the `generated_CPP/` directory, and the generated files for the TDF modules are stored under the `generated_H/` directory. The generated source code files for the software of the virtual prototype are stored under the `generated_src/` directory. The generated topcell is stored in the `generated_topcell/` directory.

```
$HOME/TTool/
├── SysCAMSGenerationCode/
│   ├── generated_CPP/
│   │   └── *_tdf.h
│   └── generated_H/
│       └── *_tdf.h
└── MPSoC/
    ├── generated_src/
    │   ├── main.c
    │   └── Block0.c
    └── generated_topcell/
        └── top.cc
```

Listing B.1: Generated code files directories.

The GPIO2VCI component was created under the `connectivity_component/` directory, as shown in Listing B.2

```
$HOME/TTool/MPSoC/soclib/soclib/module/connectivity_component/gpio2vci/caba/
└── metadata/
    ├── gpio2vci.sd
    ├── source/include/
    │   └── gpio2vci.h
    └── source/src/
        └── gpio2vci.cpp
```

Listing B.2: GPIO2VCI component directories.

Listing B.3 shows the java files that were modified or created as part of the integration of the SystemC-AMS modules and SoCLib modules into TTool.

```
$HOME/TTool/src/main/java/
├── ui/
│   ├── window/
│   │   ├── JDialogSysCAMSExecutableCodeGeneration.java
│   │   ├── JDialogSysCAMSBlockDE.java
│   │   ├── JDialogSysCAMSBlockTDF.java
│   │   ├── JDialogSysCAMSPortConverter.java
│   │   ├── JDialogSysCAMSPortDE.java
│   │   └── JDialogSysCAMSPortTDF.java
│   └── AvatarDeploymentPanelTranslator.java
├── syscamstranslator/
│   ├── toSysCAMSCluster/
│   │   ├── ClusterCode.java
│   │   ├── Header.java
│   │   ├── PrimitiveCode.java
│   │   └── TopCellGeneratorCluster.java
│   ├── SysCAMSTBlockTDF.java
│   ├── SysCAMSSpecification.java
│   ├── SysCAMSTPortDE.java
│   ├── SysCAMSTPortTDF.java
│   ├── SysCAMSTPortConverter.java
│   └── SysCAMSValidateException.java
└── ddtranslatorSoclib/
    ├── toSoclib/
    │   ├── Gpio2VciAddress.java
    │   ├── TaskFileSoclib.java
    │   └── TasksAndMainGenerator.java
    ├── toTopCell/
    │   ├── Declaration.java
    │   ├── Header.java
    │   ├── MappingTable.java
    │   ├── NetList.java
    │   ├── Platforminfo.java
    │   ├── Signal.java
    │   └── TopCellGenerator.java
    ├── AvatarAmsCluster.java
    └── AvatarddSpecification.java
```

Listing B.3: Java files created or modified for the integration of SystemC-AMS and SoCLib modules.

The `libsyscams` library created to provide interface functions for communication with the GPIO2VCI component is shown under Listing B.4.

```
$HOME/TTool/MPSoC/mutekh/libsyscams/
├── gpio2vci_address.c
├── gpio2vci_address.h
├── gpio2vci_iface.c
├── gpio2vci_iface.h
├── libsyscams.config
└── Makefile
```

Listing B.4: `libsyscams` library source files.

Listing B.5 shows other files that were modified as part of the integration tasks.

```
$HOME/TTool/MPSoC/
├── Makefile.forsoclib
└── generated_topcell/
    └── config_noproc
```

Listing B.5: Other modified files for the integration tasks.

C. Appendix: Static schedule computation of model from Section 4

To compute the static schedule of the TDF modules shown in Figure C.1, the static scheduling algorithm proposed in [25] and shown in Listing C.1 can be used.

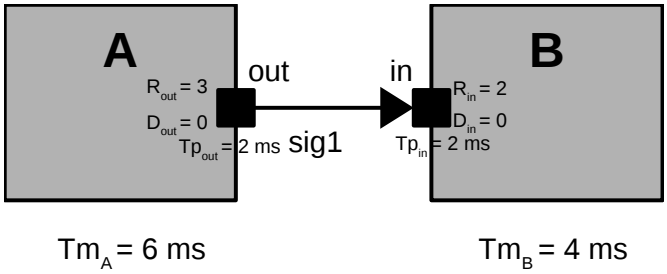


Figure C.1: TDF model from Section 4

```
1: procedure computeStaticSchedule
2:     Solve for the smallest positive integer vector q
3:     Form an ordered list L of the nodes (α) of the model.
4:     for each node α do
5:         if α is runnable then
6:             Schedule α
7:         end if
8:     end for
9:     if each node α has been scheduled q_α times then
10:        STOP
11:    else if no node could be scheduled then
12:        DEADLOCK
13:    else GO to 4
14:    end if
15: end procedure
```

Listing C.1: Sequential Scheduling algorithm (Adapted from [25]).

The topology matrix Γ for this TDF cluster is formed by one row that represents the signal `sig1` and two columns representing the modules **A** and **B** respectively. Output TDF port rates will add a positive value to an element of the matrix, while input TDF port rates will add a negative value to that element. For this TDF cluster, the matrix Γ is:

$$\Gamma = \begin{bmatrix} 3 & -2 \end{bmatrix}$$

And solving:

$$\Gamma \cdot q = 0$$

$$\begin{bmatrix} 3 & -2 \end{bmatrix} \cdot \begin{bmatrix} q_A \\ q_B \end{bmatrix} = 0$$

for the smallest positive integer vector q, we have:

$$q_A = 2$$

$$q_B = 3$$

This vector q represents the number of times that each module should be executed.

The list L of nodes will be {**A**, **B**}. Module **A** is runnable because it is not waiting for any TDF samples, so it is scheduled first, and produces 3 samples. So the schedule so far is:

A

At this point the next node **B** in the list is runnable because there are 3 samples available in its input port. So it is scheduled and consumes 2 samples. The schedule is now:

AB

Since **A** needs to be executed 2 times, and it is runnable, it can be scheduled again, producing 3 more samples. The schedule is:

ABA

The next node in the list is **B**. Since there are 4 samples available and this node needs to be executed 3 times, it will be scheduled twice. So the schedule is:

ABABB

Each node has been scheduled q times, so the scheduling algorithm is completed. The final schedule for one period of this TDF cluster is **ABABB**.

D. Appendix: TTool's usage scenario

For this usage scenario, the TDF model shown in Figure D.1 will be modeled and simulated in TTool. Module **A** will write a value of 2 to module **B**. Module **B** will read that value, multiply it by the last value received from the GPIO2VCI component, and transmit the result to the GPIO2VCI component which will be connected to the SoCLib interconnect component of an SoC platform.

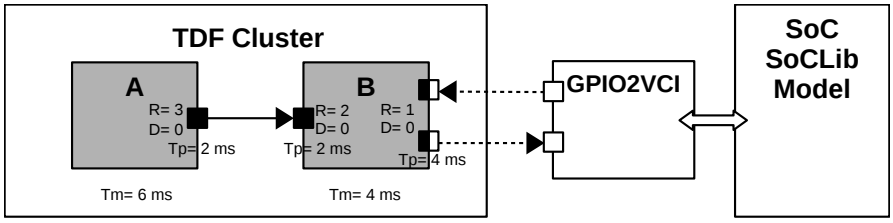


Figure D.1: TDF Cluster model

Before installing TTool, a `global.config` file should be created under `$HOME/.soclib/`. This file can found in Listing A.6. In order to install and execute TTool, run the following commands under the `$HOME/TTool/` directory:

```
> make ttool
> make install
> ./ttool.exe
```

After opening TTool, go to File>New Model. Right click on the design area and select "New SystemC-AMS Block Diagram". A new SystemC-AMS panel will open. Right click on the panel and select New SystemC-AMS Diagram. A new SystemC-AMS Component Diagram panel will open. In the same way, several SystemC-AMS Component Diagrams can be created inside the SystemC-AMS panel.

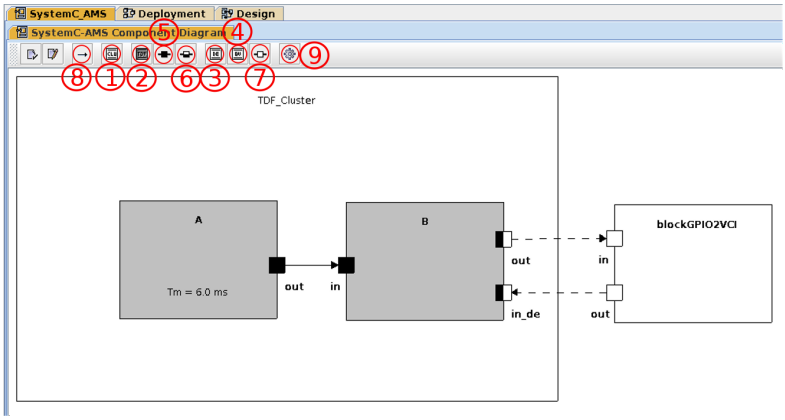
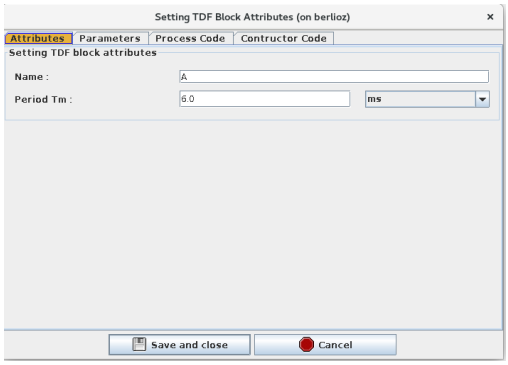


Figure D.2: TDF Cluster creation in the SystemC-AMS Component Diagram panel.

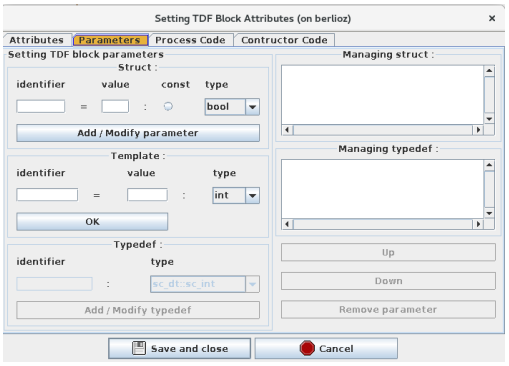
Inside the SystemC-AMS Component Diagram panel TDF clusters can be created. To create a TDF cluster click on the "Cluster" button, number 1 of Figure D.2, and click anywhere inside the SystemC-AMS Component Diagram panel to place the TDF Cluster block. Double-click to change the name of the TDF cluster. The size of the TDF cluster can be adjusted.

To add TDF module blocks, click on the "TDF Block" button, number 2 of Figure D.2, and click anywhere inside the TDF Cluster block to place the TDF Module block. To add a DE module block follow the same procedure, just start by clicking on the "DE Block" button, number 3 of Figure D.2. To add a GPIO2VCI block, click on the "GPIO2VCI block" button, number 4 of Figure D.2. GPIO2VCI blocks should be placed outside of the TDF Cluster block.

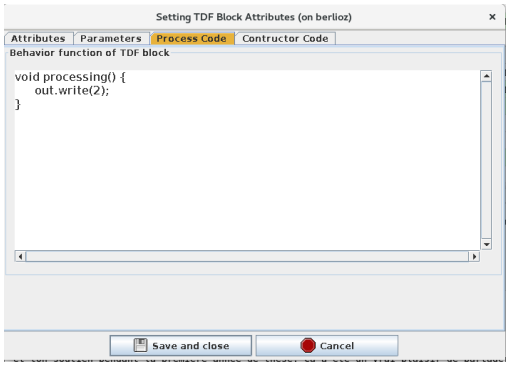
The properties of the TDF module blocks can be set by double-clicking the block. A new window will open, as shown in Figure D.3. In the Attributes panel the name and module timestep (Tm) including time units can be set, as Figure D.3a shows. In the Parameters panel, seen in Figure D.3b, the parameters of a TDF module such as its internal variables or template parameters can be also set up. In the Process Code panel, the `processing()` function of the module can be set, as Figure D.3c shows. Finally, if constructor code needs to be added, it can be done in the Constructor Code panel. The attributes of the DE module blocks can be modified in the same way. The GPIO2VCI block has no attributes to be modified.



(a) Attributes panel



(b) Parameters panel



(c) Process Code panel

Figure D.3: TDF module block attributes window.

When the required modules have been created they need to be connected through their ports. The TDF ports and converter ports can be added to the TDF module blocks. Click on the "TDF port" button, number 5 of Figure D.2, to add a TDF port. Click on the "Converter port" button, number 6 of Figure D.2, to add a TDF converter port. DE ports can be added to the DE blocks and to the GPIO2VCI block by clicking on the DE port button, number 7 of Figure D.2. The attributes of the ports can be modified by double-clicking a port, as shown in Figure D.4. The name, timestep (Tp) along with the time units, rate, delay, type and origin of the port can be modified. Note that if a TDF module or a DE module will be connected to the GPIO2VCI component, the type

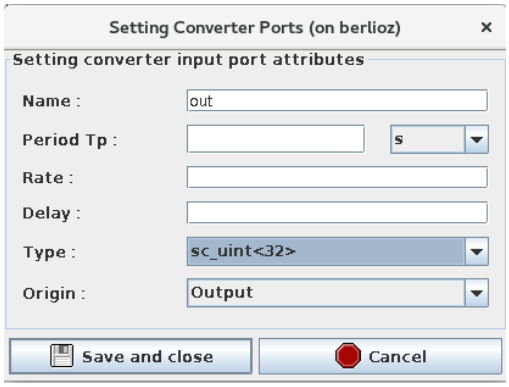
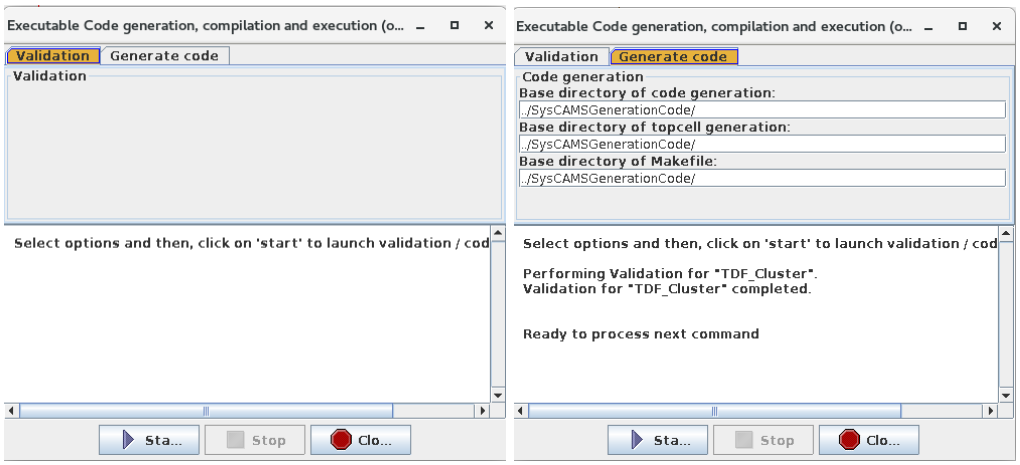


Figure D.4: Setting port attributes.

`sc_uint<32>` should be selected as shown in Figure D.4. For DE ports, the port can be added to the sensitivity list of the module by enabling the Sensitive field and selecting if the port will be sensitive to a positive or negative edge of the incoming signal or `null` for any incoming signal change. To connect the blocks, click the "Connector" button, number 8 of Figure D.2, and then click an output port to connect it with an input port.

Once a TDF cluster model has been created. The next step is to validate the correctness of the model. This is done by clicking on the "Generate SystemC-AMS code" button, as shown in number 9 from Figure D.2. This will open a new window, where validation of the model and code generation can be made. Click on the "Start" button to start the validation of the model, as shown in Figure D.5a. The Validation panel will display a message stating if there is an error with the model and make suggestions on how to fix it. If the model is valid, then a success message will be displayed and the Generate Code panel will open, as shown in Figure D.5b. Click on the "Start" button again to generate the SystemC-AMS code for the model.



(a) Validation panel **(b)** Generate Code panel

Figure D.5: Validation and code generation window.

In parallel, the Software Design and the Deployment Diagrams can be created. Right click on the tabs section of the design area and select "New Design" to create a new Software Design panel. A Block diagram can be created there, as shown in Figure D.6. Click on the "Block" button, number 1 of Figure D.6, to add a new block. Note that a new panel is created automatically, with the name of the block. Go to the **Block0**

panel. Here, state machine diagrams that allow to design the software can be created, as shown in Figure D.7. For this model, one state will be added by clicking on the "State" button, number 1 of Figure D.7, and placing it in the panel. A stop block can be added by clicking the "Stop" button, number 2 of Figure D.7. Finally the states should be connected by clicking the "Connect" button, number 3 of Figure D.7.

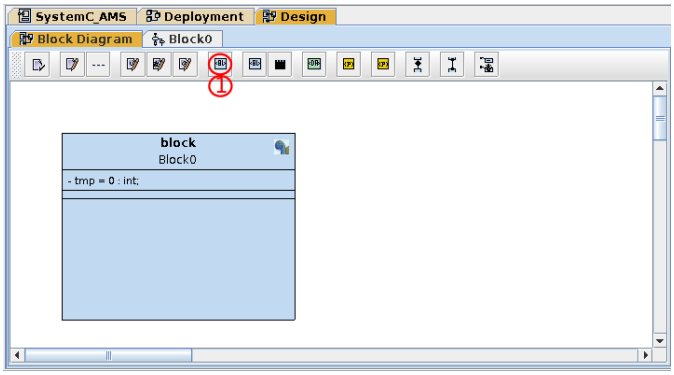


Figure D.6: Software design Block Diagram panel.

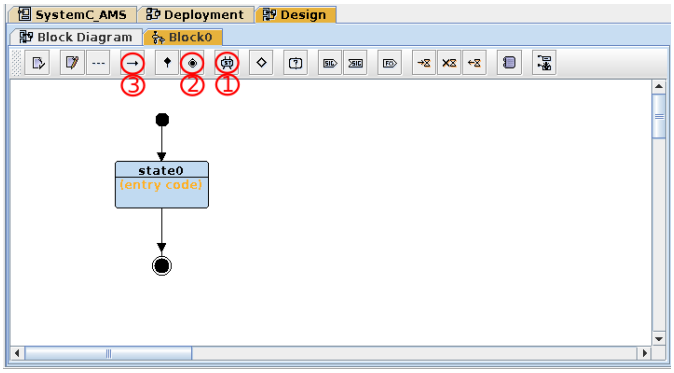


Figure D.7: Software design State Machine Diagram panel.

By double-clicking the state block, C code can be entered manually in the Prototyping tab. Here is where the functions to communicate to the GPIO2VCI component can be added as shown in Figure D.8. For the software of this model, a value of 5 will be written to the GPIO2VCI component. This value will be transmitted to the TDF cluster components. Then the output from the TDF cluster will be read and printed to the TTY component of the model. The code is shown in Listing D.1.

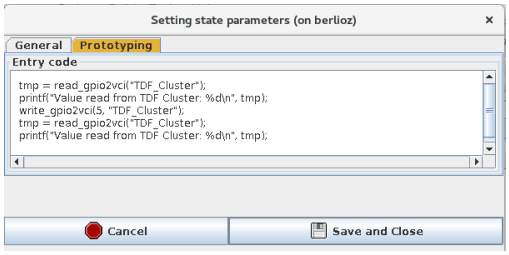


Figure D.8: State block Prototyping panel .

```
tmp = read_gpio2vci("TDF_Cluster");
printf("Value read from TDF Cluster: %d\n", tmp);
write_gpio2vci(5, "TDF_Cluster");
tmp = read_gpio2vci("TDF_Cluster");
printf("Value read from TDF Cluster: %d\n", tmp);
```

Listing D.1: State block code.

Note that the code is using a variable `tmp`. To create the variable in the Block Diagram panel, double click **Block0** to open the attributes window for the block, as shown in Figure D.9. In the Attributes panel, new variables can be added by giving an identifier name, an initial value and a type.

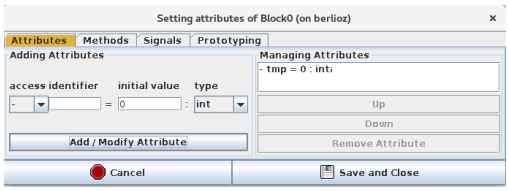


Figure D.9: Block diagram's Block attributes.

Once that the software design is complete, the MPSoC model needs to be created in the Deployment Diagram. Here, the user can insert SoCLib components and the TDF clusters. To insert a CPU click the "CPU" button, number 1 of Figure D.10. Double click the CPU block and setup the necessary attributes. To add a RAM memory click on the "RAM" button, number 2 of Figure D.10. Double click the RAM block and set up its attributes. To add a TTY console click on the "TTY" button, number 3 of Figure D.10. Finally an interconnect component needs to be added, by clicking the "VGMN" button, number 4 of Figure D.10. To map the software blocks from the Block Diagram into a specific CPU, click the "Map and AVATAR block" button, number 5 of Figure D.10, and place it under the CPU. Double click the block inside the CPU and select the name of the block that is mapped to that CPU.

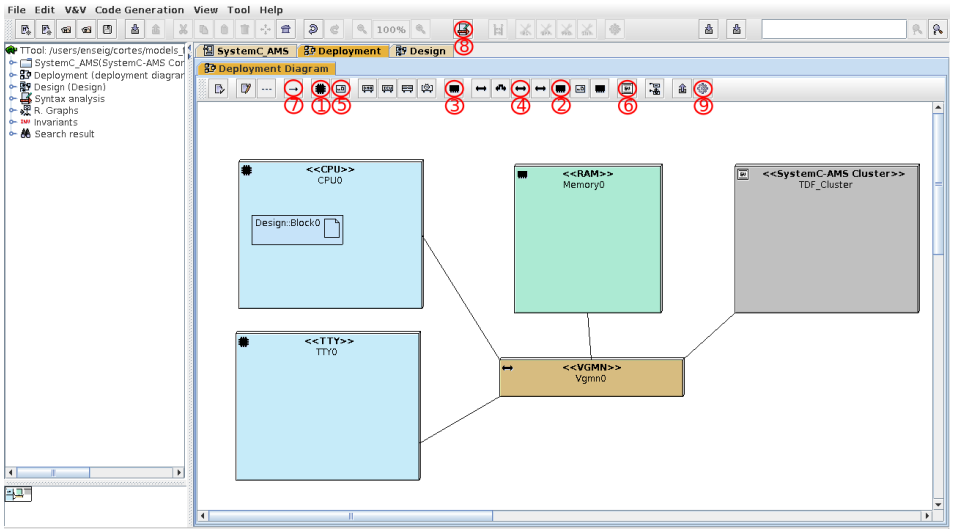


Figure D.10: Adding SystemC-AMS Clusters to the Deployment Diagram.

In order to include the TDF clusters into the MPSoC model, they need to be added as SystemC-AMS Cluster blocks in the Deployment Diagram. To add a new SystemC-AMS Cluster block, click on the "Cluster" button, number 6 of Figure D.10, and place the block in the Deployment Diagram panel. The name of the SystemC-AMS Cluster block should be the same name provided in the SystemC-AMS Component diagram. All the blocks should be connected to a SoCLib interconnect component using a connector, number 7 of Figure Figure D.10. Once the necessary SystemC-AMS Cluster blocks have been added, the topcell from the Deployment Diagram model can be generated. In the Deployment Diagram Panel, click on the "Syntax analysis" button, number 8 of Figure D.10. This will open a new window to verify the syntax of the model, as shown in Figure D.11. Click on the "Check syntax" button. If there are any syntax errors, a message will be displayed, otherwise we can proceed to the generation of the topcell.

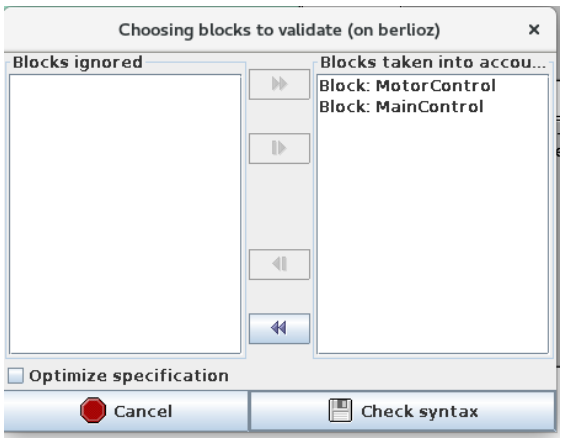


Figure D.11: Check syntax window.

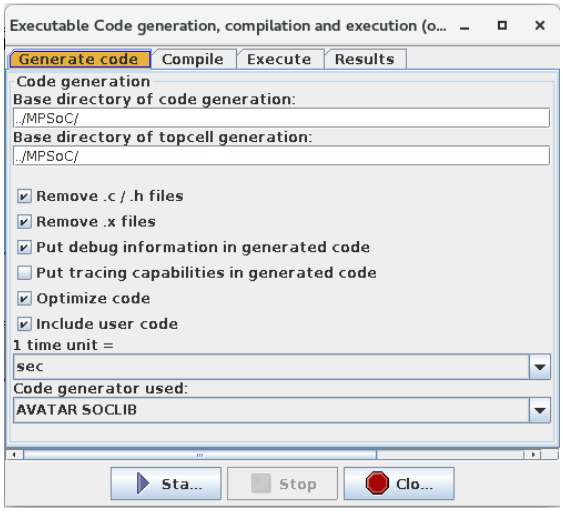


Figure D.12: Code generation window.

Click on the "Generate Deploy SoCLib" button in the Deployment Diagram panel, number 9 of Figure D.10. A new window will be opened where the topcell code can be generated, compiled and executed. In Figure D.12, the Generate Code panel is shown, where several option can be chosen, including the tracing capabilities and debugging information. Click "Start" to generate the topcell `top.cc` code and the software code.

Then in the Compile panel, click “Start” to compile the code. Finally in the Execute panel, click “Start” to begin the simulation of the virtual prototype of the model.

Figure D.13 shows the TTY console from the model. In the last lines, the values being read from the TDF cluster are printed. The first value is 0, since nothing have been written to the TDF cluster. The last value is 10, since a value of 5 was written to the TDF cluster, and it is being multiplied by the value 2 generated from the TDF module **A**.

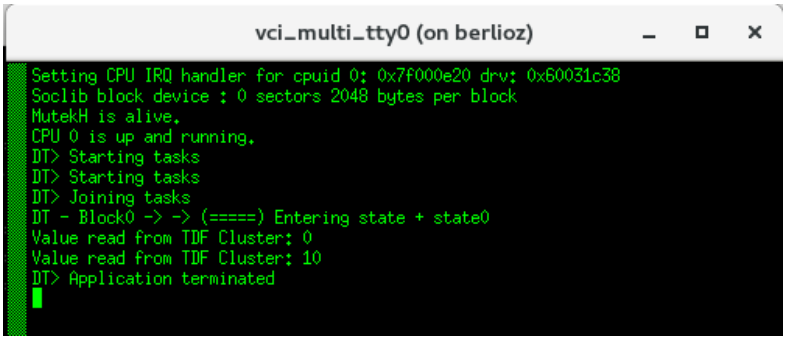


Figure D.13: Simulation output from the TTY component of the model.

E. Appendix: Lists

List of Figures

List of Tables

List of Listings

List of Abbreviations

timed-Coloured Petri Nets timed-CPN 6

Transaction Level Modeling TLM 6

Virtual Component Interface VCI 6

virtual generic micro-network VGMN 6

virtual generic system bus VGSB 6

References

[1] Päivi Parviainen, Juha Takalo, Susanna Teppola, and Maarit Tihinen. Model-Driven Development. Processes and practices. Number 114 in VVT Working Papers. VTT Technical Research Centre of Finland, February 2009.

[2] Ludovic Apvrille. TTool, an open-source toolkit for the modeling and verification of embedded systems. `http://ttool.telecom-paristech.fr/`. As of: October 4, 2018.

[3] Daniela Genius and Ludovic Apvrille. Virtual Yet Precise Prototyping: An Automotive Case Study. In *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, pages 691–700, TOULOUSE, France., January 2016. 8th European Congress on Embedded Real Time Software and Systems.

[4] Irina Lee. TTool/SystemC AMS : Interface Graphique et Génération de Plateforme. Master 1 Project. (Unpublished). Université Pierre et Marie Curie, 2018.

[5] SoCLib. `http://www.soclib.fr/`. As of: October 4, 2018.

[6] Alexandre Becoulet. MutekH. `http://www.mutekh.org/`. As of: October 25, 2018.

[7] Liliana Andrade. *Principles and implementation of a generic synchronization interface between SystemC AMS models of computation for the virtual prototyping of multi-disciplinary systems.* PhD thesis, Université Pierre et Marie Curie, 2016.

[8] CATRENE (CA701) H-INCEPTION. Heterogeneous Inception. `https://www-soc.lip6.fr/trac/hinception`, 2015. As of: October 5, 2018.

[9] Accellera Systems Initiative. SystemC. `http://www.accellera.org/downloads/standards/systemc`. As of: October 4, 2018.

[10] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, January 2012.

[11] Edward Ashford Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7(1-4):25–45, October 1999.

[12] Martin Barnasconi, Karsten Einwich, Christoph Grimm, Torsten Maehne, and Alain Vachoux. *Standard SystemC AMS extensions 2.0 Language Reference Manual.* Accellera Systems Initiative, March 2013.

[13] Martin Barnasconi, Christoph Grimm, Markus Damm, Karsten Einwich, Marie-Minerve Louërat, Torsten Maehne, François Pecheux, and Alain Vachoux. *SystemC AMS extensions User's Guide.* Open SystemC Initiative (OSCI), March 2010.

[14] Edward Ashford Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[15] Alain Greiner. Writing efficient Cycle-Accurate, Bit-Accurate SystemC simulation models for SoCLib. `http://www.soclib.fr/trac/dev/wiki/WritingRules/Caba`,

September 2017. As of: October 16, 2018.

[16] John Aynsley. *OSCI TLM - 2.0 Language Reference Manual.* Open SystemC Initiative (OSCI), July 2009.

[17] VSI Alliance. Virtual Component Interface Standard (OCB 2 2.0). Technical report, VSI Alliance, August 2000.

[18] Cédric Ben Aoun. *Principles and Realization of a Virtual Prototyping Environment for Composable Heterogeneous Systems.* PhD thesis, Université Pierre et Marie Curie, 2017.

[19] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets. Modelling and Validation of Concurrent Systems.* Springer, 2009.

[20] José-Inácio Rocha, Luís Gomes, and Octávio Páscoa Dias. Dataflow model property verification using Petri net translation techniques. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 783–788, July 2011.

[21] Liliana Andrade, Torsten Maehne, Alain Vachoux, Cédric Ben Aoun, François Pêcheux, and Marie-Minerve Louërat. Pre-simulation symbolic analysis of synchronization issues between discrete event and timed data flow models of computation. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1671–1676, March 2015.

[22] Daniela Genius, Marie-Minerve Louërat, François Pêcheux, Ludovic Apvrille, and Haralampos Stratigopoulos. Modeling Heterogeneous Embedded Systems with TTool. DUHDe 2018 - 5th Workshop on Design Automation for Understanding Hardware Designs. (Unpublished), March 2018.

[23] SoCLib tools documentation. `http://www.soclib.fr/doc/`. As of: October 18, 2018.

[24] Markus Damm, Christoph Grimm, Jan Haase, Andreas Herrholz, and Wolfgang Nebel. Connecting SystemC-AMS models with OSCI TLM 2.0 models using temporal decoupling. In *2008 Forum on Specification, Verification and Design Languages*, pages 25–30, September 2008.

[25] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[26] Stephen A. Edwards. Dataflow Languages. [PowerPoint presentation] 2001. `http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/dataflow.pdf`. As of: November 05, 2018.

[27] EchOpen project: Designing an Open Source and Low-Cost Echo-Stethoscope. `http://www.echopen.org/`. As of: November 05, 2018.