TUCS

Moazzam Fareed Niazi

A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture

Turku Centre *for* Computer Science

TUCS Dissertations
No 171, March 2014

A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture

Moazzam Fareed Niazi

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in Auditorium Beta on March 26, 2014, at 12 noon.

University of Turku
Department of Information Technology
FIN-20014 Turku, Finland

2014

Supervisors

Principal Scientist, Dr. Tiberiu Seceleanu
ABB Corporate Research
Forskargränd, 721 78 Västerås
Sweden

Professor Hannu Tenhunen
Adjunct Professor Pasi Liljeberg
Department of Information Technology
University of Turku
FIN-20014 Turku
Finland

Reviewers

Professor Detlef Streitferdt
Software Architectures and Product Lines Group
Ilmenau University of Technology
Helmholtzplatz 5, 98693 Ilmenau
Germany

Dr. Vassilios A. Chouliaras
School of Electronic, Electrical and Systems Engineering
Loughborough University
Ashby Road, Loughborough, Leicestershire, LE11 3TU
United Kingdom

Opponent

Professor Jari Nurmi
Department of Electronics and Communications Engineering
Tampere University of Technology
FIN-33101 Tampere
Finland

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

ISBN 978-952-12-3028-8
ISSN 1239-1883

Abstract

The capabilities and thus, design complexity of VLSI-based embedded systems have increased tremendously in recent years, riding the wave of Moore's law. The time-to-market requirements are also shrinking, imposing challenges to the designers, which in turn, seek to adopt new design methods to increase their productivity. As an answer to these new pressures, modern day systems have moved towards on-chip multiprocessing technologies. New architectures have emerged in on-chip multiprocessing in order to utilize the tremendous advances of fabrication technology.

Platform-based design is a possible solution in addressing these challenges. The principle behind the approach is to separate the functionality of an application from the organization and communication architecture of hardware platform at several levels of abstraction. The existing design methodologies pertaining to platform-based design approach don't provide full automation at every level of the design processes, and sometimes, the co-design of platform-based systems lead to sub-optimal systems. In addition, the design productivity gap in multiprocessor systems remain a key challenge due to existing design methodologies.

This thesis addresses the aforementioned challenges and discusses the creation of a development framework for a platform-based system design, in the context of the *SegBus* platform - a distributed communication architecture. This research aims to provide automated procedures for platform design and application mapping. Structural verification support is also featured thus ensuring correct-by-design platforms. The solution is based on a model-based process. Both the platform and the application are modeled using the *Unified Modeling Language.* This thesis develops a *Domain Specific Language* to support platform modeling based on a corresponding UML profile. *Object Constraint Language* constraints are used to support structurally correct platform construction. An emulator is thus introduced to allow as much as possible accurate performance estimation of the solution, at high abstraction levels. VHDL code is automatically generated, in the form of "snippets" to be employed in the arbiter modules of the platform, as required by the application. The resulting framework is applied in building an actual design solution for an MP3 stereo audio decoder application.

Tiivistelmä

Sulautettujen järjestelmien tarjoama laskentakapasiteetti, niiden koko mitattuna transistorien määrällä sekä kompleksisuus ovat kasvaneet huomattavasti viimevuosien aikana Mooren lakia mukaillen. Näiden laitteiden tuotantovaatimukset ovat myös kiristyneet tehden suunnitteluprosessista entistä vaativamman suunnittelijoille. Tämä on pakottanut etsimään yhä uusia suunnittelumenetelmiä jotta pystytään vastaamaan kiristyneihin tuotantovaatimuksiin. Eräs vastaus näihin vaatimuksiin on ollut siirtyminen moniprosessorijärjestelmiin. Tämä teknologia puolestaan tuo uusia arkkitehtuuriasi haasteita ja mahdollisuuksia hyödyntää teknologisen kehityksen tuomia mahdollisuuksia.

Sovellusalustapohjainen suunnittelu on eräs ratkaisu näihin haasteisiin. Tämän paradigman kantavan ajatuksena on eriyttää kyseessä olevan sovelluksen funktionaalisuus ja järjestelmän sisäinen tiedonsiirto useilla eri abstraktio tasoilla. Olemassa olevat sovellusalustapohjaiset suunnittelumenetelmät eivät ole täysin automatisoituja kaikilla suunnitteluprosessin osaalueilla. Lisäksi järjestelmän sisäinen tiedonsiirto aiheuttaa ongelmia tässä prosessissa, sillä useasti järjestelmät eri osat eivät ole täysin yhteensopivia.

Tässä väitöskirjatyössä pyritään ratkaisemaan yllämainittuja ongelmia sekä esitetään kehitetty sovellusalustapohjainen sulautettujen järjestelmien suunnittelumenetelmä. Menetelmää on sovelluttu hajautettuun SegBus kommunikaationa arkkitehtuuriin. Tähän liittyen työssä on kehitetty automatisoituja menetelmiä sekä alustan suunnitteluun että sovellusten vaatimien resurssien sijoitteluun kyseiseen järjestelmään. Menetelmä tukee strukturaalista oikeellisuuden varmistamista joka perustuu mallipohjaiseen lähestymistapaan. Objektipohjaista konseptia on sovellettu tukemaan strukturaalisesti oikeellista rakenteen varmistamista. Työssä on kehitetty myös emulaattori, jonka avulla on mahdollista arvioida tarkasti suorituskykyä. Työssä kehitetty sovellusalustapohjaista menetelmää on sovellettu audiojärjestelmän dekoodaukseen.

To the memory of my beloved mother (1958-2007)

Acknowledgments

First praise is to ALLAH, the Almighty, on whom ultimately we depend for sustenance and guidance. Without His help and blessings, I was not able to bring this work to its completion.

The completion of this doctoral thesis is an exciting moment of my life and I am feeling glad while writing this part of my thesis. I would like to express my gratitude to several people and institutions that have influenced the research work presented in this thesis.

First and foremost, I would like to express my heartfelt gratitude to my first two supervisors, Adjunct Professor *Tiberiu Seceleanu* and Professor *Hannu Tenhunen*, for their guidance, encouragement and support that they have provided to me during the course of this work. Especially, I wish to thank Dr. *Tiberiu Seceleanu* for considering me for this research work, and for all the support and guidance that he provided to me on numerous occasions in Turku as well as in Västerås. I have always experienced the true inspiration from him every time we discussed the ongoing work in a much friendlier environment.

I would also like to extend my gratitude to my third supervisor Adjunct Professor *Pasi Liljeberg* for comprehensive guidance throughout my doctoral studies, and especially for handling the practical matters during the last phase of this work. I am also very thankful to Dr. *Dragoş Truşcan* and Dr. *Torbjörn Lundkvist* from Åbo Akademi for providing me guidance related to UML modeling and tools in the start of this research work.

I would also like to thank Professor *Detlef Streitferdt* from Ilmenau University of Technology, Germany and Dr. *Vassilios A. Chouliaras* from Loughborough University, United Kingdom for the detailed reviews and constructive feedback on this thesis. Furthermore, I wish to sincerely thank Professor *Jari Nurmi* from Tampere University of Technology, Finland for accepting to be the opponent in the public defense of my thesis.

I gratefully acknowledge the financial support from different institutions that made this research work possible. I am grateful to Academy of Finland and Turku Centre for Computer Science (TUCS) for providing me the financial support during the initial four years of this research work. I

wish to express my sincerest gratitude to TES-Tekniikan edistämissäätiö and University of Turku Graduate School (UTUGS) for their financial support through incentive grants on two different occasions.

I owe my deepest gratitude to my friend Dr. *Qaisar Malik* for all the support, help and kindness that he has provided to me while he was completing his doctoral studies in Åbo Akademi. I wish to appreciate him for everything that he has done for me during all those years we spent together in Turku. Similarly, I would also like to take this opportunity to thank my oldest friend *Ali Hanzala Khan* for providing me a pleasant company during all those years which we have spent together in Pakistan, Sweden and Finland as well. He always showed me that there is a life outside the academic environment. Moreover, I am also grateful to *Syed Asad Jafri*, *Ali Shuja Siddiqui*, *Fareed Ahmed Jokhio*, *Adnan Ashraf* and many others for providing me a good company during my stay in the ICT House, and also to all others for exchanging smiles and saying greetings while passing through the department corridors.

I would also like to extend my gratitude to my teachers from my former university (Bahria University, Pakistan): Dr. *Muhammad Mohiuddin*; Dr. *Syed Aley Imran Rizvi*; and *Salman Zafar.* Their continuous motivation throughout my post-graduate studies kept my mind fertile, productive and focused to my career goals.

I would also like to thank all the secretaries especially *Irmeli Laine, Maria Prusila, Heli Vilhonen, Päivi Rastas*, administrative and support staff especially *Tomi Mäntylä* and *Sami Nuuttila*, both at TUCS and Department of Information Technology of the University of Turku for running the departments in an efficient and hassle-free manner.

Most importantly, none of this would have been possible without the best wishes, love, prayers and patience of my family. I am truly indebted to my mother for everything that she has done for me. Although she is no longer with us but she will be remembered for the rest of my life. I also gratefully acknowledge the support and motivation from my father and all my brothers, especially the elder ones. I am truly thankful to all of them.

Finally, I am truly grateful for the all the support, understanding and love from my wife throughout the time we spent together while completing this research work and writing this thesis as well. I appreciate her for all the unconditional support, patience and nice advices that she provided to me in difficult times.

Turku, March 2014
Moazzam Fareed Niazi

"Seek knowledge from the cradle to the grave."
Prophet Muhammad (Peace and blessings be upon him)

Contents

List of Figures

List of Abbreviations

μs	micro second
ACG	Automatic Code Generation
AM	Application Model
API	Application Programming Interface
BU	Border Unit
CA	Central Arbiter
CAD	Computer-Aided Design
CIM	Computation Independent Model
CoD	Complexity of Design
CoPE	Correctness of Platform Execution
CPM	Complete Platform Model
CWM	Common Warehouse Metamodel
DE	Designer Expertise
DocA	Distributed on-chip Architecture
DSL	Domain Specific Language
DSM	Domain Specific Modeling
EDA	Electronic Design Automation
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FU	Functional Unit

GI	Graphical Interface
HDL	Hardware Description Language
HLDE	High-Level Design Entry
IP	Intellectual Property
M2M	Model-to-Model transformation
M2T	Model-to-Text transformation
MDA	Model Driven Architecture
MDD	Model Driven Development
MOF	Meta-Object Facility
MP3	MPEG-1/MPEG-2 Audio Layer III Encoding Format
MPEG	Moving Picture Expert Group
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
OCL	Object Constraint Language
OOP	Object-Oriented Programming
OS	Operating System
P-SDF	Package SDF
PAM	Partitioned Application Model
PC	Personal Computer
PE	Platform Emulation
PIM	Platform Independent Model
PO	Performance Optimization
PSM	Platform Specific Model
RTL	Register Transfer Level
SA	Segment Arbiter
SAM	Segmented Application Model

SDF	Synchronous Dataflow
SoC	System-on-Chip
SoPC	System-on-Programmable Chip
SPM	Synthesizable Platform Model
SysML	Systems Modeling Language
TCT	Total Clock Ticks
UML	Unified Modeling Language
UP	Useful Period
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuit
WP	Waiting Period
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

An *embedded system* is a special-purpose computer which is generally part of a larger physical system and provides a dedicated, and sometimes programmable, functionality to its ambient environment. Nowadays, such systems are widely used in airplanes, automobiles, industrial systems, cameras, medical equipment, house-hold appliances and inside so many other application areas.

In recent years, the complexity of the embedded systems has increased tremendously, along with the shrinking silicon features. The time-to-market is also shrinking, imposing challenges for the designers to adopt new design methods.

As an answer to the new capabilities and pressures, embedded computing systems have moved towards on-chip multiprocessing architectures. These include on-chip multiprocessing to utilize the tremendous advances of fabrication technology. The Distributed on-chip architectures (DocA) [20] or multiprocessor system-on-chip (MPSoC) [4] paradigm have gained increasing support from system designers. MPSoC is seen as one of the foremost means through which performance gains are still to be sustained even after Moore's law expected demise [27]. The most common DocA / MPSoC platforms are *network-on-chip* (NoC) [1], and *segmented bus* platforms [16][20].

As the complexity of the application is increasing with time, the designers are facing software and hardware challenges when designing applications targeting MPSoCs. However, it is also a challenge to take full benefits from all the distinguished features provided by the MPSoC platforms such as programability, enormous potential for performance gains, multiple clock domains, scalability, heterogeneity support, parallelism, etc. At the moment, there are a number of difficulties in MPSoC development and the prominent one is the lack of comprehensive design methodologies that can deal with almost every aspect of the design processes [27].

The platform-based design approach [3][33][34] provides the means to address these challenges. Few important features of this approach include: fast time-to-market, programability, short design cycle, design reusability (both hardware and software), design customization, huge potential for optimizations (performance, power consumption, etc.). The principle behind that approach is to separate the functionality of an application from the structural organization and communication architecture of a hardware platform at several levels of abstraction. This not only eases the design process but also delivers efficiency for concurrent engineering in specific aspects of the system by different members of the design team. The essential step in this approach is the mapping process where functions of the application to be implemented are assigned (mapped) to the components of the hardware platform spread within a single chip or on a number of chips. Targeting initially single chip designs, the approach gains even more importance in the context of DocA / MPSoC.

The existing design methodologies don't provide full automation in every level of the development process, and sometimes, the co-design of platform-based systems lead to sub-optimal systems. In order to offer an optimum match, platform specific characteristics must be taken into consideration for each application.

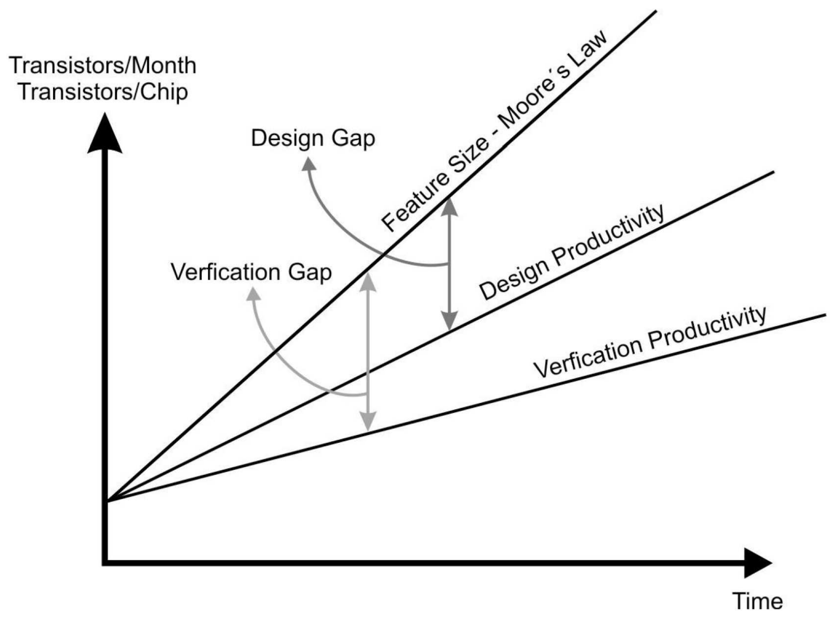


Figure 1.1: Comparison of gaps among other important trends of SoC design [18].

Moreover, the *design productivity gap* in MPSoC design remains one of a key challenge in the existing design methodologies. This challenge can be addressed by developing new *computer-aided design* (CAD) tools/frameworks based on modern design methods while improving the reusability of IP components. This not only enables the designers to take full advantages of MPSoC platforms, but further satisfies other important constraints e.g. time-to-market, quality of results, costs, etc. Figure 1.1 compares the design productivity gap with the advances in silicon technologies and with the growth trends in *design verification* [18]. Here, the gap between the chip complexity (Moore's law) and design productivity is increasing with the passage of time, which is important to realize now and must be addressed by the *Electronic Design Automation* (EDA) community.

Design decisions taken at higher abstraction levels are expected to have the highest impact on the quality of the system implementation. Given the complexity of modern on-chip multiprocessor solutions, this seems to gain even more in importance. On the other hand, optimality of design is strictly affected by platform parameters; hence, such platform-level aspects, if taken into account at high abstraction levels will lead to a solution that maximizes the benefits of the distinguished features offered by the MPSoC platform. The specific platform considered in this study is the *SegBus* platform [20], which is an *on-chip communication platform* architected to provide communication infrastructure between the connected IP-cores in an MPSoC environment.

The three basic and most commonly used approaches of the design methodologies used for the embedded system design are: *top-down*; *meet-in-the-middle* [2]; and *bottom-up*. The *top-down* methodology begins with a high-level system specification which specifies initially a system structure with required elements. Next, the specification is iteratively refined by decomposing system into sub-systems and components until enough details are gathered for each component. The design is then partitioned into hardware and software parts, and co-optimized. Hence, the approach is also well-known as *hw/sw co-design*. On the other hand, a *bottom-up* approach is an inverse approach where, firstly, individual components are built, and integrated afterward to eventually develop a target system.

In the *meet-in-the-middle* [2] approach, the application(s) and the platform are developed independently. The platform is developed by employing a *bottom-up* design flow where the low-level aspects of the platform are considered, whereas the application is developed by following a top-down flow starting from the high-level aspects of the application. When both specifications are complete, the application is then mapped on to the platform and this is the distinguished feature of the meet-in-the-middle approach. This approach is heavily dependent on a comprehensive library of software and hardware elements which reduces the design time and efforts and improves reusability. However, the establishment of such a library requires significant efforts beforehand.

In this thesis, the work is highly influenced by the meet-in-the-middle approach because of two important reasons. First, the thesis considers a particular platform (the *SegBus* platform), which is intended to be used as a communication platform. A number of applications from the same family can be mapped on that platform, and thus those applications can be executed in parallel. Second, the platform promotes the reuse of *intellectual property* (IP) components at different levels of abstraction which addresses other important complexity challenges e.g. design cost, time-to-market, etc.

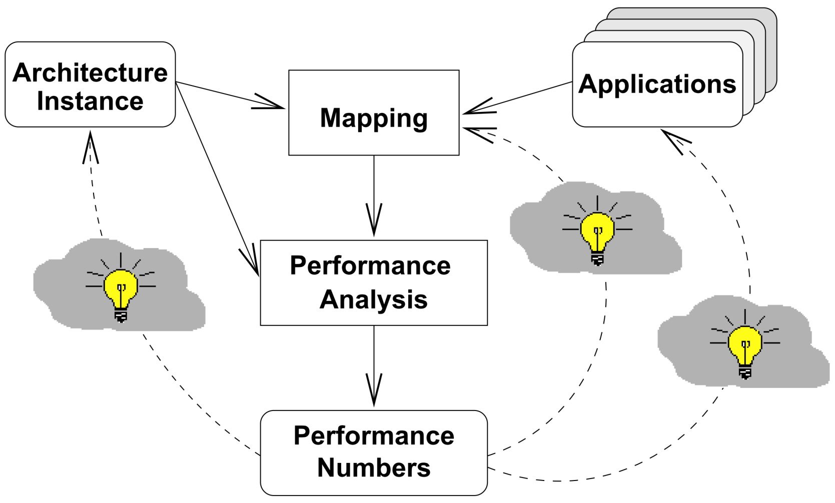


Figure 1.2: The Y-chart approach [30].

1.1 The Y-chart approach

The *Y-chart* [29][30] is a generic and iterative design space exploration methodology for platform-based designs. The methodology is based on the meet-in-the-middle approach where the application and platform are developed independently but the platform is fine-tuned *iteratively* to achieve the performance required by that application(s) at a certain level of abstraction.

In this methodology, an instance of the platform is known as a *configuration*. In the *mapping* process, the mapping of the application onto a platform configuration is performed. The general overview of this approach is depicted in Figure 1.2. The artifacts (application(s), platform/architecture and mapping) are equally important to achieve an optimal configuration and thus, performance level. The performance aspects of the mapped application running on a platform configuration are examined in *Performance*

Analysis step and these *performance numbers* determine the efficiency of chosen configuration. If the configuration is not optimal, then the designer get another chance to further explore the *design space* to produce a new configuration and the process continues iteratively. This way, the author can truly exploit all the essential parameters of the employed platform in order to achieve the target performance goals.

1.2 The Model-Based Paradigm and UML

Software-based computing systems have been around for a few decades now. Their increasing complexity with time is a known challenge to all those associated with their design. At the beginning, it was relatively easy to design and implement such software systems with some low-level programming languages such as FLOW-MATIC, COBOL, Autocode, etc. But as the time went by, system specifications became more difficult for newer generation of software systems and designers realized that abstraction levels had to be raised. Consequently, high-level programming languages like C, FORTRAN and Pascal were developed and employed in the 1970s. At the time, these high-level programming languages were not competent enough to tackle increasingly complex specifications, and as a result, the *object-oriented* paradigm came into existence. The paradigm was based on a simple idea - *Everything is an object.* Later, the object-oriented paradigm gained popularity among designers and larger systems were developed by employing this paradigm [5]. However, the ever increasing complexity of such systems approached the limits of object-oriented paradigm and a new paradigm introduced: the *model driven* paradigm. The paradigm is based on a simple idea: *Everything is a model.* In fact, the idea is not so new because it has been used previously in database systems.

The software engineering community gradually adopted the model based paradigm, and as a result, the system specification moved from textual description (C, C++, Java, etc.) to visual/graphical *modeling languages*, which provide a higher-level of abstraction in modeling environments for system specification. The Unified Modeling Language (UML) [65] is one of them.

A *notation* is a graphical entity, and basically, forms the syntax of a modeling language. UML is a general-purpose modeling language which includes a set of graphical notations which are used in a particular diagram (e.g., class diagram, state diagram, etc.) for specifying various aspects of a software system in a generic manner. It can be applied to a variety of application domains, most importantly to object-oriented and component-based systems. Furthermore, the UML's *extensibility mechanism* (i.e., *stereotypes*, *tagged values* and *constraints*) enables the introduction of new building blocks from the existing ones. This capability opens the door to raising the abstraction level further - a modeling area, known as, *domain specific modeling* (DSM). For DSM, UML is extensively applied initially to build *domain specific language* (DSL) based on an associated UML *profile* targeted for a particular domain (discussed briefly in Chapter 4).

The paradigm shift from programming languages to modeling languages, as discussed earlier, for system specification has introduced a new methodology for software engineering known as the Model Driven Architecture (MDA) [61] promoted by the Object Management Group (OMG). The primary focus of MDA is on *models*, *automation* (by *model transformations*) and *code generation* which in turn, shifts the targeted system's design concerns from low-level implementation to high-level solution modeling. UML is the key enabling technology for MDA, and thus, it is the default modeling language for MDA. The model transformation is a key feature in the MDA approach. It enables the transformation of a model from one abstraction level to another, and hence, is the driving force which shifts models from higher to lower-levels of abstraction, and finally towards implementation. The important kinds of model transformation are: model-to-model (M2M), and model-to-text (M2T). The thesis employs the model-to-text (M2T) transformation in its intended approach, as it is briefly discussed in Chapter 5.

In a networking session [28] a few years ago, on-chip MPSoCs were identified as the preferred architecture for future complex embedded system design. Still, as discussed earlier, lack of right tools, design frameworks and methodologies didn't allow EDA vendors to support it as the mainstream architecture for embedded systems implementations. This thesis intends to take a small step further by making use of UML and MDA, towards developing a unitary framework for design and verification of applications targeting an MPSoC platform.

1.3 Design and Verification Challenges

A few of the challenges in designing applications targeting DocA (in general) and *SegBus* platform (in particular) addressed here in this thesis are as follows.

CoD - complexity of design

In the context of embedded system design, the term *complexity* refers to amount of uncertainty to move from one phase of design process to another. Embedded systems are inherently complex [52]. This is particularly true for modern embedded systems executing complex software applications and always demanding new features. The system complexity further increases as the number of specific design constraints increase. Some of these design constraints includes:

- real-time constraints relating to particular functions of the system.
- crucial time-to-market constraints that become essential for the success or failure of system under development.
- power consumption constraints which are always important for battery-operated systems.
- low-cost constraints which decide the commercial viability of system.
- endless pressure for increasing functionality.

The above mentioned and many other design constraints play a key role in the immense growth of system complexity. One way of dealing with complexity is with the use of *models* and *model-based development* (MBD) (briefly elaborated in Chapter 3.2). Generally, the word 'model' refers to describe any given system aspect with *simplicity.* Here, simplicity refers to hiding details which are less important pertaining to a certain concern, but those details could be important ones in some other models when concern changes. The process of building models is called *modeling.* At the moment, modeling plays a significant role in systems engineering (especially core software systems). The modeling process is based on *divide-and-conquer* approach where the designer limits his/her focus only to certain aspects of the system-under-development. Different members with specific expertise from the development team then concentrate on particular aspects of the system and provide solutions in a quick manner. This bring the thesis to *its* definition of the term '*model-based development*' that the thesis follows, because there can be numerous interpretation of the generic term. Model-based development or MBD is a development approach with the use of models which are specified in a particular domain specific language, relating to a specific application domain. MBD has been used in different application domains [62][63] to minimize development effort and time.

PO - performance optimization
This is an important, and most of the times a primary challenge in embedded system design. In recent years, it has got foremost attention especially in digital signal processing (DSP) applications. Consequently, it plays a major role in the increased CoD. New multiprocessing architectures have been introduced to deal with CoD and PO. An intelligent *mapping* of application functions on the multiprocessing

architectures is key to exploit tremendous features (including performance) from such architectures. A *bad* mapping mostly makes the chosen architecture a sub-optimal and unnecessary cost(s).

The approach for PO, which is discussed in this thesis, various mappings and allocation scenarios will provide different performance results, given the inter task communication requirements both at application and at platform levels.

CoPE - correctness of platform execution
This deals with the proper ordering of application tasks after allocation, considering the support for parallelism.

DE - designer expertise
Following CoD, experienced developers may be required in the process, possessing detailed knowledge about platform characteristics and how issues may be solved at platform levels.

1.4 Thesis Approach and Contributions

This thesis presents an approach to addressing the previously mentioned challenges. The thesis presents a model-based development and verification framework for embedded systems in general, and MPSoC in particular. As mentioned earlier, the particular MPSoC platform that the thesis considers is the *SegBus* platform. From a high-level perspective, the key contributions of this thesis are structured around the following directions.

HLDE - high level design entry
The thesis uses the *Unified Modeling Language* (UML) for high-level design entry for both the platform (*SegBus*) and to support the allocation and mapping of application tasks on that platform. By providing the HLDE solution, the thesis addresses the CoD, CoPE and DE challenges, while also providing the necessary support for the solutions detailed below.

PE - platform emulation
Emulation is used here to provide estimative results prior to having an actual implementation of the application on the platform. This is an important step in observing at early design stages the quality and performance of design. Addresses PO, CoPE and DE.

ACG - automatic code generation
A general solution to address CoD and DE. In the present context, it also addresses CoPE.

The *Unified Modeling Language* (UML) [65] has been utilized in novel design methods proposing a solution for the challenges in the design of complex electronic systems. Here, the purpose is to provide a unified environment for platform modeling, application mapping and system emulation, such that performance is estimated and adjusted to optimal levels in a correct and efficient manner. The key contributions of this thesis are briefly listed below.

1. The thesis introduces **model-based development** (MBD) support for *SegBus* design. The choice of MBD helps the designers to address CoD (by HLDE), CoPE (support for structured design, and correctness related aspects - customization classes, OCL constraints, etc., support for design reuse - library). It also addresses DE, as a hardware non-specialist (such as a software designer) may also develop applications without being fully aware of the specifics of the underlying hardware platform.

   The thesis then presents a technique to create a **graphical interface** (GI) in the form of *SegBus* DSL within an existing modeling tool [60] to leverage modeling infrastructure for the analysis of various *SegBus* instances that may answer to specific application requirements. The customization of each platform element is defined in the form of user-defined rules. These customization rules set properties on each profile element about their relationships, ownerships (briefly discussed in Chapter 4.2). The customization rules impose restrictions on profile elements during application / platform modeling, in order to provide a structurally correct version of the platform instance.

   The GI answers especially the CoD and CoPE challenges, providing the HLDE solution. By including model validation features, the GI also serves the DE challenge.

2. The thesis develops an *emulator* program targeting the *SegBus* platform as part of the framework to perform emulation on the modeled configuration in DSL. A code generation engine transforms the platform-independent and platform-dependent models of application, as modeled in DSL, into XML schemes. The generated XML schemes are then employed by the emulator program to estimate the utilization of platform elements with respect to data transfers and total application's execution time. After the analysis of the emulation results, the designer is able to take decisions on whether the emulated configuration is the best/optimal or not for the target application, and can fine-tune platform configuration before moving towards lower levels of the design process.

Hence, the realization of the emulator addresses, via PE the PO, CoPE and DE challenges.

3. The thesis presents methods to generate the application's execution schedule from system models which are used later to implement the communication platform's arbiters. Such capability is developed within the emulator program. If designer finds the performance aspects from the emulated system at an optimum/best level, the designer can generate the "application dependent" arbiter-level control code based on VHDL [41], in an automated manner.

   Hence, with this technique, the thesis addresses the CoPE and DE challenges via GI, PE and ACG.

1.5 Related Work

As it has been discussed in section 1, the design methodologies and frameworks being used today for embedded systems development lack automation in many stages of their design process. While building the proposed framework, several research studies have been studied where the authors present concepts in particular areas of embedded system design. For instance, few studies, which have come across, deal with high-level modeling of the embedded systems while others express ideas specifically for system emulation. Following are listed few studies which are similar to the proposed framework up to some extent.

The UML Profile for MARTE [69] has been standardized by OMG a few years ago. As the name suggests, MARTE is intended for the modeling and analysis of real-time and embedded systems in a generic way. The profile consists of various packages and notations, which help to specify and validate target systems at a number of abstraction levels. However, MARTE does not provide a methodology for designing any particular systems (real-time and embedded) [48]. A lack of a proper methodology, for mapping applications on a particular platform (*SegBus* platform in the context of this thesis), gets prime focus when considering prospective improvements in the designer productivity. Gamatié et al. [22] introduced *Gaspard* framework for designing massively parallel embedded systems. The framework is based on MARTE profile where system is defined initially, and refined in a step-by-step manner for final implementation. Since their framework does not target to any specific implementation platform (unlike this thesis's approach which targets the *SegBus* communication platform), therefore it provides them opportunities to explore a comparatively wider design space. The refinements in their framework are based on model-driven principles, which are similar to the work presented in this thesis in many aspects.

Moreover, Architecture Analysis & Design Language (AADL), standardized by SAE International [70], is a modeling language for software-intensive system, which has gained popularity especially in safety-critical systems in recent years. However, the issues highlighted in [23] makes it less attractive for use in the proposed framework.

Kangas et al. [21] introduced a UML-based framework named as *Koski* for MPSoC design and implementation. Koski provides tools ranging from application modeling to architecture implementation. The framework is based on a custom UML profile (the *TUT profile*) similar to this research. The framework also supports synthesis and automated design space exploration for identifying the right architecture for the target application. At the same time, their approach does not consider a definite architecture/platform for application implementation, unlike this work which focused on the *SegBus* platform. Another difference compared to this approach is that they do not employ MDA and package-based communication in their approach as this research do since the communication is a primary design concern for modern MPSoC systems. Despite that their design flow seems to be inspired by the MDA principles.

R. Thomson et al. [49] developed and implemented a H.264 decoder unit in an UML environment, exposing models for both hardware and software components. The *FalconML* tool is used to model different aspects of the applications via various UML diagrams, and the flow eventually produces RTL code (VHDL) which can be used to synthesize the design. The difference to the approach of this thesis is that this research work considers a target platform for implementation whereas their approach generates a customized solution.

Many DSLs have been proposed targeting a variety of application domains and this research considers the *SegBus* platform as the backbone of the embedded MPSoC system. Consel et al. [32] introduced the *Spidle* - DSL for specifying streaming applications and a compiler for generating source code for the intended software. The approach was validated experimentally by comparing the source code generated by Spidle compiler with an equivalent C source code. A number of optimizations in the Spidle compiler are missing, such as, temporal and spatial locality in data and instruction caches, the performance impact of input streams buffering, etc. Similarly, Arora et al. [31] presented a DSL for introducing an application-level *Checkpointing and Restart* (CaR) mechanism in heterogeneous and distributed environments such as computational grid. The idea is to make sequential and parallel system fault-tolerant by introducing code for the CaR mechanism in high-level specifications.

Riccobene et al. [53] presented a UML profile for SystemC and defined a language to specify, analyze, design, and visualize different artifacts of the SystemC language in a SoC design flow. These SystemC artifacts includes, for example, SystemC primitive channels (signal, mutex, semaphore, etc.), SystemC core language artifacts (modules, ports, processes, etc.), SystemC data types (bits, bit vectors, 4-valued logic type, etc.). The profile provides a modeling framework for systems in which high-level models can be refined down to an implementation language. However, the work concluded that there is still a need to develop appropriate mechanisms and tools to fully utilize UML-based profiles system development with automation support. Unlike their work, this research proposes a comprehensive UML-based profile for the *SegBus* platform embedding structural semantics in a graphical form while imposing restrictions among graphical elements at the same time.

Seceleanu et al. [19] introduced *SBTool* - a tool for handling optimum device allocation on the *SegBus* platform based on the criteria to minimize and balance traffic loads between devices across the platform. This work takes advantage of the *SBTool* in the design methodology to get estimates of the optimal placement of different devices over specified number of bus segments.

Truscan et al. [50] introduced a model-based design methodology for implementing applications on the *SegBus* platform. Their methodology allows both application and platform to be modeled independently in earlier phases, and after refinements in both models (application and platform), the application can be mapped onto the platform in a stepwise manner. The proposed methodology has been inspired by their methodology in many aspects and the proposed methodology has been further evolved and customized for the objectives of this thesis.

The primary objective in designing system emulator is to obtain as much as possible accuracy in estimating the execution characteristics that can be expected from a real platform. Several research studies have been presented in recent years where the target was to develop emulators for different hardware platforms, specially for the Network-on-Chip (NoC) platform [1], but there exists a number of emulation tools for other areas as well.

Jaber et al. [7] presented early architecture exploration methodology based on DIPLODOCUS framework [46][47] to examine the effects of shared resources (CPU, bus, memory, etc.) on a system's performance metrics such as throughput, latency and resource utilization at high levels of abstraction. Their methodology also facilitates the modeling of interactions of the system with its environment. Their architecture modeling approach divides the architecture into 3 classes of nodes: *computations nodes* (CPUs, DSPs, hardware accelerators, etc.); *communication nodes* (buses, routers, switches, etc.); and *storage nodes* (shared memories). They employed a library of pre-defined abstract models of different nodes (limited only to architecture) which is similar to this thesis's approach of employing a library of reusable IP components. On the contrary, the proposed framework's library can also store the application-related functional components which is dissimilar to their library approach. Further, they introduced a SystemC-based simulation environment for monitoring and analyzing the modeled system. This is comparatively similar to the proposed emulator in this thesis which is also used for monitoring and analyzing the utilization of platform elements with respect to data transfers and execution time. In addition, their study is focused only to architectural exploration and analysis of simulation results, and the comparisons with actual execution results over real architecture setup are missing, unlike to this thesis's approach which compares the emulated execution results with the real ones for the sake of predicting emulator's accuracy.

Schelle et al. [54] introduced an emulation tool - *NoCem*, for NoC exploration. The tool provides the capability to emulate memory architectures, asymmetric processor configuration, special purpose offload engines, etc. The tool calculates path latencies in clock cycles, used for any particular transfer between processor cores and provides a detailed view of the communication bottlenecks within the NoC platform. This is fairly similar to the platform emulator developed in this framework, which is used to emulate a particular configuration of *SegBus* platform. Performance bottlenecks can be discovered and then can be adjusted.

Liu et al. [39] presented *NoCOP* - an emulation and verification framework for exploring on-chip interconnection architectures. An instruction-set simulator (ISS) and universal serial bus (USB) communicator (the terms mentioned in their paper) have also been introduced. The ISS and USB communicator execute on a host computer and are used to set the parameters of the emulation environment. Through the experimental results using both software and hardware, the authors proved that the proposed emulation/verification framework can speed up the simulation of the network-on-chip (NoC) architecture, and decrease resource usage when targeting Field Programmable Gate Array (FPGA) silicon. The design under emulation needs to be programmed onto a FPGA device. A separate host computer is responsible for initializing and managing emulation of the programmed design in the FPGA. This makes it less flexible compared to the proposed approach which is more flexible and does not require any FPGA device and consideration about deeper levels of abstraction in the early stages of the design process.

Genko et al. [24] presented a NoC emulation platform implemented on FPGA. The NoC hardware platform has been implemented on a Virtex-II Pro FPGA, which consists of network injection, reception and controller components. The integrated PowerPC processor core functions as a controller. Instead of merely being the platform where the circuit is prototyped, the method can speed up functional validation and add flexibility to the NoC configuration exploration. The major drawback in their approach is the use of FPGA device as a prototyping platform with the purpose of predicting performance scores, unlike the proposed approach in this thesis, which does not require any prototyping platform for performance estimation.

MDA has been utilized in different design areas to provide automation in design activities up to some extent [IV]. MDA provides a set of guidelines which designers use to specify the system requirements in a structured manner. The resulting system specifications are expressed as models. With the refinement techniques, the designer can ultimately converge the system models toward a complete system with the help of the transformation techniques (Model-to-Model, Model-to-Text, etc.) applied to different models. Below, a few studies are listed where the authors employ MDA in embedded system development at different abstraction levels.

Vidmantas et al. [26] introduced MDA methods where the designer can create "Platform Independent Model" (PIM) of an application using UML together with SysML plug-in. They introduced techniques to transform PIM into a PSM model, which is later transformed into textual source code specifically for one operating system (OS). The authors have considered more than one OS where the modeled application can be run, unlike the proposed approach where there is no consideration of OS is required. Further, this thesis discusses PIM (known as P-SDF in this thesis) and PSM models of any target application similar to their approach, but this thesis does not generate source code of actual application's functionality which is dissimilar to their approach. Instead the thesis generates the control code for different arbiters of the *SegBus* communication platform, which supports application execution on the platform and eliminates the need of an OS.

Koudri et al. [35] presented a design flow for System-on-Chip/System-on-Programmable Chip design based on the use of UML and dedicated profiles. They supported the use of the Model-Driven Development (MDD) for hardware-software co-design with an example of *Cognitive Radio Application*, implemented on FPGA. The modeling tool they used generates thousands of lines of code for the modeled example application but further improvements need to be done.

M. Thompson et al. proposed a highly automated framework titled as *Daedalus* for system-level architectural exploration, system-level synthesis, programming and prototyping of heterogeneous MPSoC platforms [44][45]. Their framework allows to construct customizable MPSoC platforms from a library of pre-defined and pre-verified IP components, similar to the proposed approach of having a library of pre-built functional components. At the same time, their approach is dissimilar as this research is closely linked to a communication platform - the *SegBus* platform.

1.6 Research Publications

During the course of conducting this research, the following articles have been published in journal and international conference proceedings. Hence, the thesis is fully or partially based on each of them.

I. Moazzam Fareed Niazi, Khalid Latif, Tiberiu Seceleanu, Hannu Tenhunen. "A DSL for the SegBus Platform", in Proceedings of the 22nd IEEE International System-on-Chip Conference (SOCC), pp. 393-398, 2009, Belfast, United Kingdom.[1].

II. Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "An Emulation Solution for the SegBus Platform", in Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS), pp. 268-275, 2010, Oxford, United Kingdom.

III. Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "A Performance Estimation Technique for the SegBus Distributed Architecture", in Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW), pp. 89-98, 2010, San Diego, USA.

IV. Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "An Automated Control Code Generation Approach for the SegBus Platform", in Proceedings of the 23rd IEEE International System-on-Chip Conference (SOCC), pp. 199-204, 2010, Las Vegas, USA.

V. Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "Towards Reuse-based Development for the On-Chip Distributed SoC Architecture", in Proceedings of the 36th Annual IEEE Computer Software and Applications Conference Workshops (COMPSACW), pp. 278-283, 2012, Izmir, Turkey.

VI. Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "A Development and Verification Framework for the SegBus Platform", Journal of Systems Architecture, vol. 59, nr. 10, part C, pp. 1015-1031, 2013. `http://dx.doi.org/10.1016/j.sysarc.2013.07.005`

1.7 Summary of the Research Publications

Paper I - Moazzam Fareed Niazi, Khalid Latif, Tiberiu Seceleanu, Hannu Tenhunen. "A DSL for the SegBus Platform", in Proceedings of

[1]The above papers have been referred in this thesis many times with a usual citation referring style e.g., [I], [II], [III], [IV], etc.

the 22nd IEEE International System-on-Chip Conference (SOCC), pp. 393-398, 2009, Belfast, United Kingdom.

High-level design entry is essential for the designing and verification of systems at higher levels of abstraction. In this paper, a *Domain Specific Language* (DSL) for a multi-core segmented bus platform - *SegBus*, is presented. The DSL, based on a UML profile, consists of graphical platform elements in the form of stereotypes with the necessary tagged values depicting platform aspects at high levels of abstraction. Customizations are applied to each stereotyped element in the form of user-defined rules to restrict relationship between platform elements. The *Object Constraint Language* (OCL) is employed to introduce structural constraints, in order to impose structural requirements between platform elements and mechanisms were introduced to validate them during system-level modeling. The paper presents a simplified example of a H.264 video encoder application where the DSL is used to specify and validate the application and platform models in a step-by-step unified way.

The author's contribution: The author has been responsible for all related work in this publication including profile development, customization of platform's elements inside the modeling tool, specification of the structural constraints, DSL viability in the problem domain, and writing the manuscript.

Paper II - Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "An Emulation Solution for the SegBus Platform", in Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS), pp. 268-275, 2010, Oxford, United Kingdom.

This paper presents an emulation solution for a multi-core segmented bus platform, SegBus, to assess the performance of any specific application on a particular platform configuration, modeled in UML. The paper presents method to transform a Platform Specific Model (PSM) of the application into *Java* source code using a modeling tool and the generated code is utilized by the emulator program to get the execution results. The solution enables the estimation of the performance on different platform configurations at early stages of the design process.

The author's contribution: The author has been responsible for all related work in this publication including writing Java source code for each of the platform elements, emulator development, performance estimation and analysis, and writing the manuscript.

Paper III - Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "A Performance Estimation Technique for the SegBus Distributed Architecture", in Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW), pp. 89-98, 2010, San Diego, USA.

The approach presented in this paper is a modified and enhanced approach of performance estimation to that discussed in paper II. Here, the paper proposes a performance estimation technique for a multi-core segmented bus platform, SegBus. The technique enables the assessment of the performance of any specific application on a particular platform configuration, modeled in Unified Modeling Language (UML). The paper presents methods to transform Package Synchronous Data Flow (P-SDF) and Platform Specific Model (PSM) models of the application into *Extensible Markup Language* (XML) schemes using a modeling tool. The generated XML schemes are utilized by the emulator program to get the execution results. The technique facilitates the performance estimation of the application mapped on a number of different platform configurations during the early stages of the design process.

The author's contribution: The author has been responsible for all related work in this publication including enhancing DSL to model P-SDF models, emulator development, development of XML parser inside the emulator, performance estimation and analysis, and writing the manuscript.

Paper IV - Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "An Automated Control Code Generation Approach for the SegBus Platform", in Proceedings of the 23rd IEEE International System-on-Chip Conference (SOCC), pp. 199-204, 2010, Las Vegas, USA.

The transition from higher abstraction levels towards an implementation in an automatic manner is very important when addressing the embedded system design productivity challenge. This paper presents a model-based approach for the generation of low-level control code for the (segment- and central-level) arbiters of the *SegBus* communication platform in order to support application implementation and scheduled execution. The approach considers Model-Based Development (MBD) as a key to modeling the application at two different abstraction levels, namely as Package-Synchronous Dataflow and Platform Specific Model, using the SegBus platform's DSL. Both models are transformed into XML schemes and then utilized by an emulator to generate the "application-dependent" VHDL code, the so-called "snippets". The obtained code is inserted in a specific section of the platform arbiters.

The paper presents an example of a simplified MP3 stereo audio decoder where the methodology is employed to generate the control code of the arbiters.

The author's contribution: The author has been responsible for all related work in this publication including writing the engine for generating the control code from the P-SDF model, embedding the developed engine in the emulator, code generation and analysis of its accuracy, and writing the manuscript.

Paper V - Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "Towards Reuse-based Development for the On-Chip Distributed SoC Architecture", in Proceedings of the 36^{th} Annual IEEE Computer Software and Applications Conference Workshops (COMPSACW), pp. 278-283, 2012, Izmir, Turkey.

Reusability in the platform-based design approach is key to dealing with increasing embedded systems design complexity. In this paper, the development of a library of reusable components for a multi-core segmented bus platform, the *SegBus*, is presented. The library is based on a plug-in that was developed and deployed within a modeling tool and used by the *SegBus* DSL for developing applications targeting the *SegBus* platform. The steps required in building the library and embedding it into a plug-in are discussed together with its use in the proposed design methodology.

The author's contribution: The author has been responsible for all related work in this publication including plug-in development, plug-in integration with the *SegBus* DSL in the modeling tool, establishing components list in the library, and writing the manuscript.

Paper VI - Moazzam Fareed Niazi, Tiberiu Seceleanu, Hannu Tenhunen. "A Development and Verification Framework for the SegBus Platform", Journal of Systems Architecture, vol. 59, nr. 10, part C, pp. 1015-1031, 2013. `http://dx.doi.org/10.1016/j.sysarc.2013.07.005`

This paper integrates all the ideas been presented in previous papers to form a distinct framework for modeling and verification. Here, the paper describes the creation of a development framework for a platform-based design approach, in the context of the distributed architecture of the *SegBus* platform. The work aims to provide automated procedures for platform build-up and application mapping. Structural verification support is also featured. The solution is based on a model-based process. Both the platform and the application are modeled using the Unified Modeling Language (UML). A Domain Specific Language (DSL) is developed to support the platform modeling, based on a corresponding UML profile. *Object Constraint Language* (OCL) constraints, in the form of a validation suite, are employed to support a structurally-correct platform construction. The DSL helps designers to correctly model application and platform in a fast manner and further helps in model transformation at later stages of development process. The validation suite embedded in the DSL helps the designer to rectify the structural problems in models and correct them with necessary measures. An emulator is consequently introduced to allow an as much as possible accurate performance estimation of the solution, at high levels of abstraction. VHDL code is automatically generated, in the form of "snippets" to be employed in the arbiter modules of the platform, as pertaining to the application specification. The resulting framework is applied to building actual design solutions for a MP3 audio decoder application. The emulation-based solution enables analyzing any platform configuration with respect to performance figures. Based on emulation results, it's the job of the designer to decide which configuration will be best suited for the final implementation. Such decisions made in the early stages of design process improve the quality of eventual system in terms of performance.

The author's contribution: The author has been responsible for all related work in this publication including DSL analysis and development in the modeling tool, construction of validation suite, emulator development, design and development of the control code generator in the emulator, verification of the presented methods on an actual example application, and writing the manuscript.

1.8 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, the thesis provides a short description of the *SegBus* platform with its structural characteristics. Next, in Chapter 3, the thesis discusses the proposed design methodology that employs the proposed model-based framework while designing applications targeting the *SegBus* platform. The development of the *SegBus* DSL and the associated library of reusable components are described in Chapter 4. The approach of emulating and estimating the modeled system is presented in Chapter 5, while the method of execution schedule generation from the modeled system is briefly discussed in Chapter 6. Furthermore, in Chapter 7, the thesis formally employs and validates the proposed framework on a real application as an example, followed by a discussion of the proposed framework. Finally, the thesis concludes in Chapter 8.

1.9 Thesis Navigation

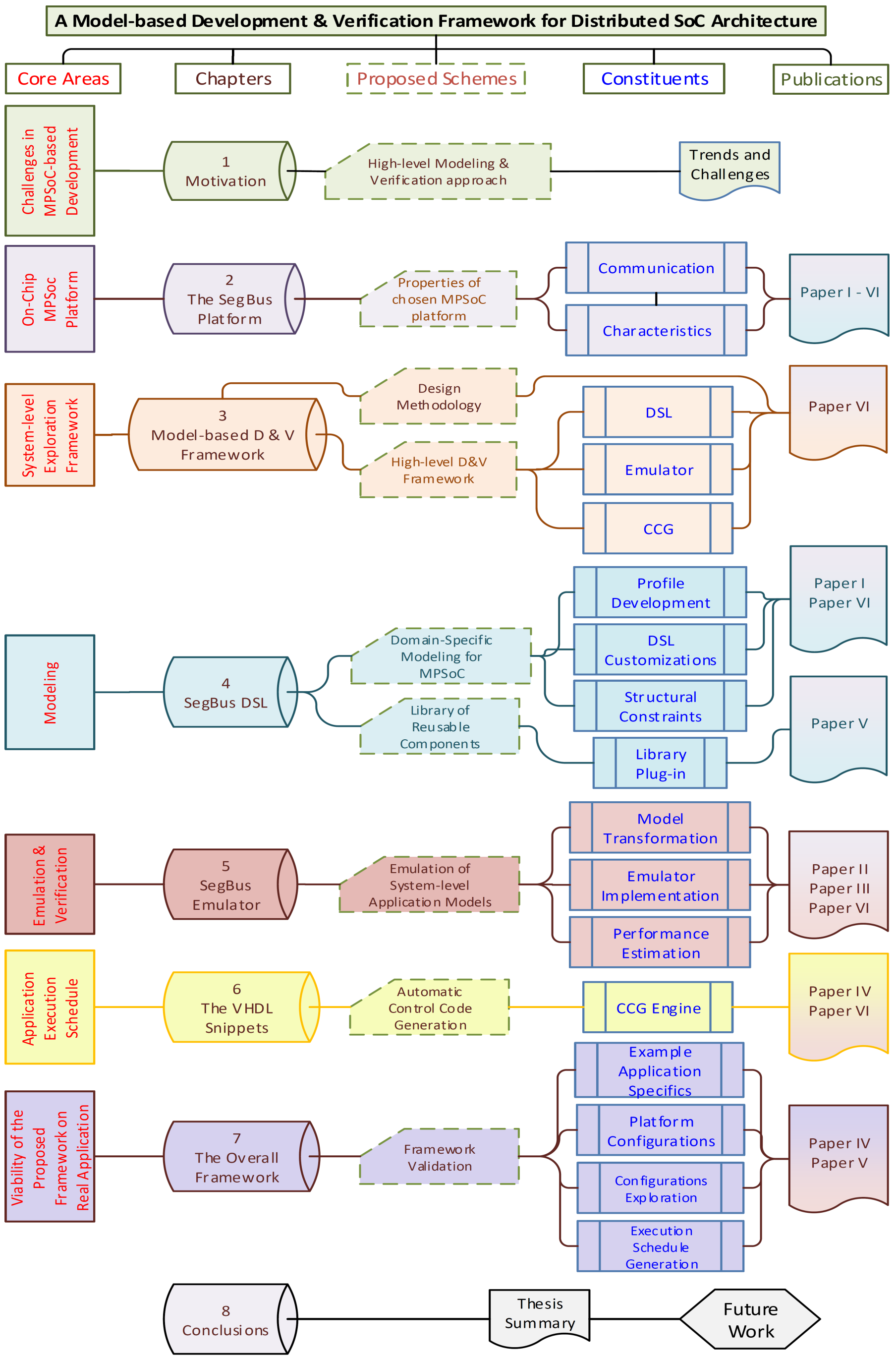


Figure 1.3: Navigation of the thesis.

Chapter 2

The SegBus Platform

A bus-based computing system comprises of a variety of modules, each with particular functionality. Each module communicates with other modules through a shared bus which connects with every module. A data exchange between these modules is termed a *bus transaction* which is administered by a set of specific rules known as *transaction protocol*. The modules are generally classified as: *masters*, *slaves* and *arbiters*.

Master refers to a module which initiates a transaction on the bus and remains active for the duration of transaction. In contrast, slave refers to a passive module which is basically the “target” of a particular bus transaction and in return, it delivers a service to requesting master. In principle, one master and possibly many slaves can be involved in a bus transaction. As the bus is a shared communication link, only one master can be allowed to own the bus at any time, that is, transfer data to or from the slave module. Hence, an arbiter does *arbitration* based on a specific priority scheme between masters on the bus. Figure 2.1 represents high-level view of a traditional bus-based computing system.

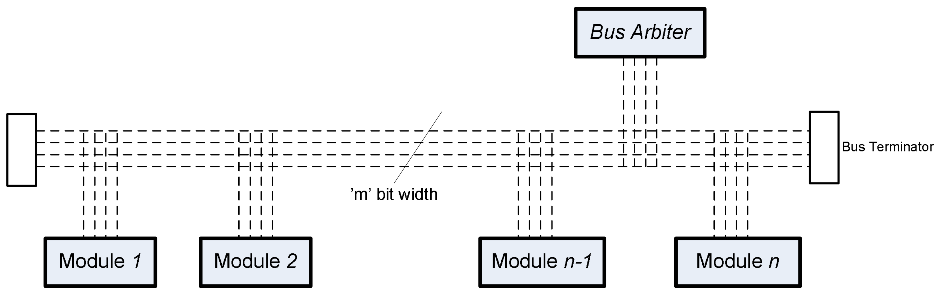


Figure 2.1: Traditional bus-based system.

The maximum bus speed is largely limited by the physical length of the bus and the number of modules connected on the bus. A lengthy bus introduce *transmission delays* in bus *lines* or *wires* due to physical effects. The major physical effects on the bus wires includes: *capacitance*, *resistance* and *impedance*, which put a restriction on the maximum bus frequency. Additionally, the number of modules connected on the bus should be within the maximum limit as defined by the bus to operate properly. Failure to consider these two important concerns result in performance degradation, increased power consumption, etc.

The bus arbiter follows a certain priority scheme for granting the bus among different masters. The priority scheme can be either *fixed priority* or *rotating priority* (also known as *round-robin scheme*). In the fixed priority scheme, the priority of each module remains the same all the time irrespective of how many times a master has been granted the bus; while in rotating priority scheme, the priority of each module changes based on the usage history. Both priority schemes have certain advantages and disadvantages and their selection is purely based on their suitability for the target system.

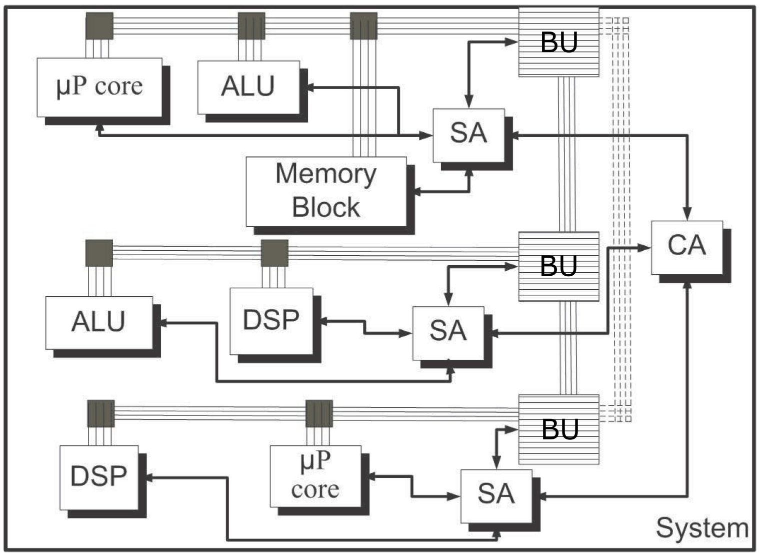


Figure 2.2: Segmented bus structure [20][50].

A segmented bus is a "collection" of individual buses (segments), interconnected with the use of FIFO structures. Each segment acts as a normal bus between modules that are connected to it and operates in parallel with other segments. Due to segmentation, each segment executes at a certain clock frequency independent from the clock rate of the neighboring segments. Therefore, the platform can be seen as a set of buses operating in parallel with different clock frequencies. Neighboring segments can be dynamically connected to each other to establish a connection between modules located on different segments. In this case, all dynamically connected segments act as a single bus during the duration of transaction. Due to the segmentation of the bus lines, and their relative isolation, parallel transactions can take place, thus increasing the performance. The *passive* segments, which are not involved in transactions, are isolated from the rest of the bus. Hence, the *active* segments, the ones that currently transfer data, offer faster operation along with lesser energy consumption per transferred bit. A high level block diagram of the segmented bus system is illustrated in Figure 2.2.

The *SegBus* communication platform [20] is built of components that provide the necessary separation of segments - *Border units* (***BU***), arbitration units - the *Central Arbiter* (***CA***) and local, *Segment Arbiters* (***SA***). The application then is realized with the support of (library available) *Functional Units* (***FU***).

The *SegBus* platform has a single ***CA*** unit and several ***SA***s, one for each segment. In addition, each segment is composed of a group of masters, a group of slaves and physical wires for data, address and control signals of the bus. A ***BU*** connects two neighboring segments. A particular segment with its connected units (masters, slaves, ***SA***) acts as standalone bus operating in parallel with other segments, where masters primarily requesting services from slaves within the same segment. The ***SA*** of each bus segment decides which device (***FU***), within the segment, will get access to the bus in the following burst transfer. At times, a master may ask for services from a slaved resident in a remote bus segment. In this case, the local ***SA*** forwards the request to ***CA*** in order to perform inter-segment communication.

The ***CA*** stores information about the current status of each segment. This includes which segments are currently involved in an inter-segment communication and which new inter-segment communication requests are still pending. In every polling cycle, the ***CA*** decides, based on this information, which pending request will be qualified to grant ownership for inter-segment communication, and eventually, the ***CA*** activates certain control signals in the relevant segments. This way, the *SegBus* platform can be seen as a *centralized*, two-level arbitration platform. At the first-level, local arbitration is used inside each segment, while at the second-level, central arbitration is employed to handle inter-segment communication requests from different ***SA***s. In this case, the ***CA*** treats individual ***SA***s as *virtual* masters requesting services from other segments.

2.1 Platform Communication

A *package* is the basic unit of communication and storage allocation in the *SegBus* platform. Within a segment, data transfers follow a "traditional" package-based bus protocol, with ***SA***s arbitrating access to local resources. The inter-segment communication follows also a package-based, circuit switched approach, with the ***CA*** having the central role. The interface components between adjacent segments and the ***BU***s, are basically FIFO elements with some additional logic, controlled by the ***CA*** and the neighboring ***SA***s. The inter-segment communication is highly dependent on the utilization of ***BU***s which transfer data in both directions. The FIFO buffer inside each ***BU*** accommodates as many data items as specified by the package size (including package header). The size of a package is not relevant from the operational point of view of the *SegBus* platform. However, larger package sizes require more chip area in each of the ***BU***s which could increase overall power consumption [20]. A brief description of the communication is given below.

Whenever one ***SA*** recognizes that a request for a data transfer targets a module outside its own segment, it forwards the request to the ***CA***. The latter identifies the target segment address and decides which segments need to be dynamically connected in order to establish a link between the initiating master and targeted slave device(s). When this connection is ready, the initiating device is granted the bus access, and it starts filling the buffer of the appropriate ***BU*** with the package data. Following a signaling protocol, the data is taken into account by the corresponding next segment ***SA***, which forwards it further, towards the destination. At this point, the ***SA*** of the targeted segment routes the package to the own bus from where it is collected by the targeted device.

A transfer from the initiating segment k to the target segment n is shown in Figure 2.3. The segments from k to n are released for other possible inter-segment operations in a cascaded manner, from the source k to the destination, n.

The *SegBus* platform employs a simple round-robin arbitration [51] at ***SA*** level, and a similar one at the ***CA*** level. The arbitration at ***CA*** level implements the application data flow, with respect to these transfers. Hence, one has to implement accurate control procedures for inter-segment transfers, as possible conflicting requests must be appropriately satisfied, in order to reach performance requirements and to correctly implement applications.

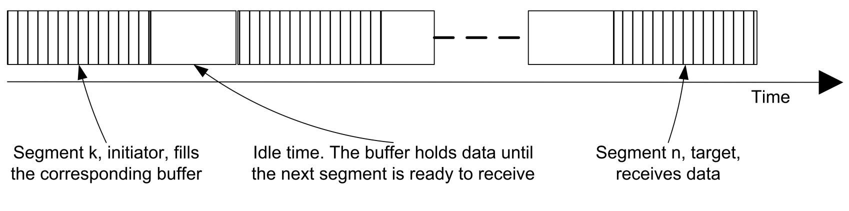

Time
Segment k, initiator, fills the corresponding buffer
Idle time. The buffer holds data until the next segment is ready to receive
Segment n, target, receives data

Figure 2.3: Inter-segment package transfer.

Both ***SA*** and ***CA*** strictly operate on the execution schedules, which are provided and implemented by the designer in each of the platform arbiters. Each entity (program line) of the execution schedules define, for instance, the source (master) and destination (slave(s)) address of data transfer request, amount of packages to be transferred, destination segment's address, etc. (briefly discussed in Chapter 6). Based on this information, the ***SA*** controls, in each grant activity, the amount of data to be transferred on bus lines by means of a local counter. Whenever the defined limit (package size) is reached, the ***SA*** clears the grant signal. This scheme eliminates the possibility of any deadlock to occur between masters and slaves. The acquisition of the bus segment can only be possible by each of them (masters and slaves), if it is specified in the related ***SA***'s execution schedule.

2.2 Platform Characteristics

The optimum performance from the *SegBus* platform can be achieved by examining 3 of its explicitly-defined global parameters: *topology*, *number of segments* and *package size*. A short description of each of them is given below.

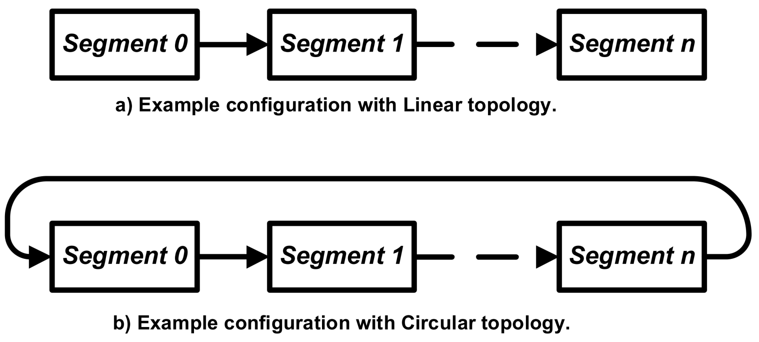

Segment 0
Segment 1
Segment n
a) Example configuration with Linear topology.
Segment 0
Segment 1
Segment n
b) Example configuration with Circular topology.

Figure 2.4: Difference between linear and circular topologies.

2.2.1 Topology

The *topology* defines the connectivity of the platform's segments: either *linear* or *circular*. In linear topology, the segments are connected in a linear fashion where each segment connects only with the adjacent segments (one left and one right) except the first and last segments which connects to only one adjacent segment (either right or left). In the circular topology, the geometry is similar to linear topology except the last segment always connects to the first segment and it creates a cycle between segment organization and thus called circular. This organization is sometimes appropriate [20] as per performance requirements. The ***CA*** determines the shortest path between any two segments in the circular configuration. Figure 2.4 shows the difference between the two topologies.

2.2.2 Number of Segments

As the name implies, this parameter sets the desired number of segments in a platform instance. Increasing the number of segments sometimes results in improved performance however this is not always true in every case. It may bring unnecessary communication overhead which affects the overall performance. Hence, increasing the number of segments to achieve a certain performance level must consider possible parallel operations in the target application(s). For interested readers, a detailed analysis of this issue can be found in [20].

2.2.3 Package Size

The unit of communication in the *SegBus* platform is called a *package*. A package consists of a header and the data load (actual data to be transferred). The header further consists of two fixed-size fields: destination address and source address. Thus, the size of the data payload influences the overall package size and therefore has to be defined initially while specifying the platform's parameters. The designer needs to consider many important factors while deciding the best/optimum size for the package. For instance, for larger packages, more data can be transferred in one transaction however, this requires larger FIFOs inside all the ***BU***s on the communication path resulting in increased chip area. A comprehensive discussion about such issues can be found in [20].

2.3 Platform Constraints

Designing a specific instance of the *SegBus* platform useful for one or more applications must consider certain constraints. These constraints are essentially the structural characteristics of the *SegBus* platform. The designer must consider such structural constraints when designing a platform instance. A few of these constraints are described as follows.

- The platform must have either a linear or a circular geometry. The topology impacts on how the "terminal" segments are connected to each other (discussed in Chapter 2.2.1).

- Every platform instance has a unique ***CA***.

- Every segment has a unique ***SA***.
- Every ***SA*** is connected to at most two ***BU***s.
- Every ***BU*** is connected to at most two ***SA***s.
- Every segment contains at least one ***FU***.
- Every ***FU*** is characterized by an own unique ID.

A second group of constraints relates to the optimal placement of modules on the platform, taking into account the communication flow described in section 2.1. Hence, appropriate control procedures must be devised such that two masters could not obtain the bus grant simultaneously. At the same time, the opportunities for parallel processing and transfers must be maintained. This can be achieved by an appropriate mechanism for the request granting. The efficiency of that mechanism must be assessed at early stages.

2.4 Summary

This chapter presented an overview of the *SegBus* platform, its architecture and characteristics. The chapter also discussed possible topologies of the platform along with a short description on the significance of the number of segments and package size. The important structural constraints and communication mechanism were also described. The work to be presented in this thesis is based on this specific MPSoC platform and the terminologies pertaining to the platform are heavily used in the rest of the thesis.

Chapter 3

The Model-based Development and Verification Framework

This chapter describes the essence of this thesis - the model-based development and verification framework. The chapter starts with a brief description of the proposed design methodology and important cornerstones of it. In addition, the chapter presents the principles of the model driven paradigm and their adoption is proposed in support of the proposed framework.

3.1 Design Methodology

This section defines the cornerstones of the design methodology utilizing the *SegBus* as a communication platform. This section briefly describes the notions used in the following sections while also presenting the existing body of work that supports the proposed approach.

Figure 3.1 illustrates a general overview of the *SegBus* design process employing a model-based framework. The proposed approach considers as the start an application model (AM) which is transformed into a partitioned application model (PAM) based on the application's core functionality and taking help from the available library elements. Each library element has a structure shown in Figure 3.5, containing a pointer to a VHDL file that describes the related module's functionality. The AM-to-PAM transformation helps to derive right granularity level of the target application, which to be implemented on an instance of the platform.

The *communication matrix* is the specification of device-to-device transactions between application components [IV][50]. Each entity in the communication matrix describe how many data items need to be transferred from one device to any other device. The matrix is generated from the PAM.

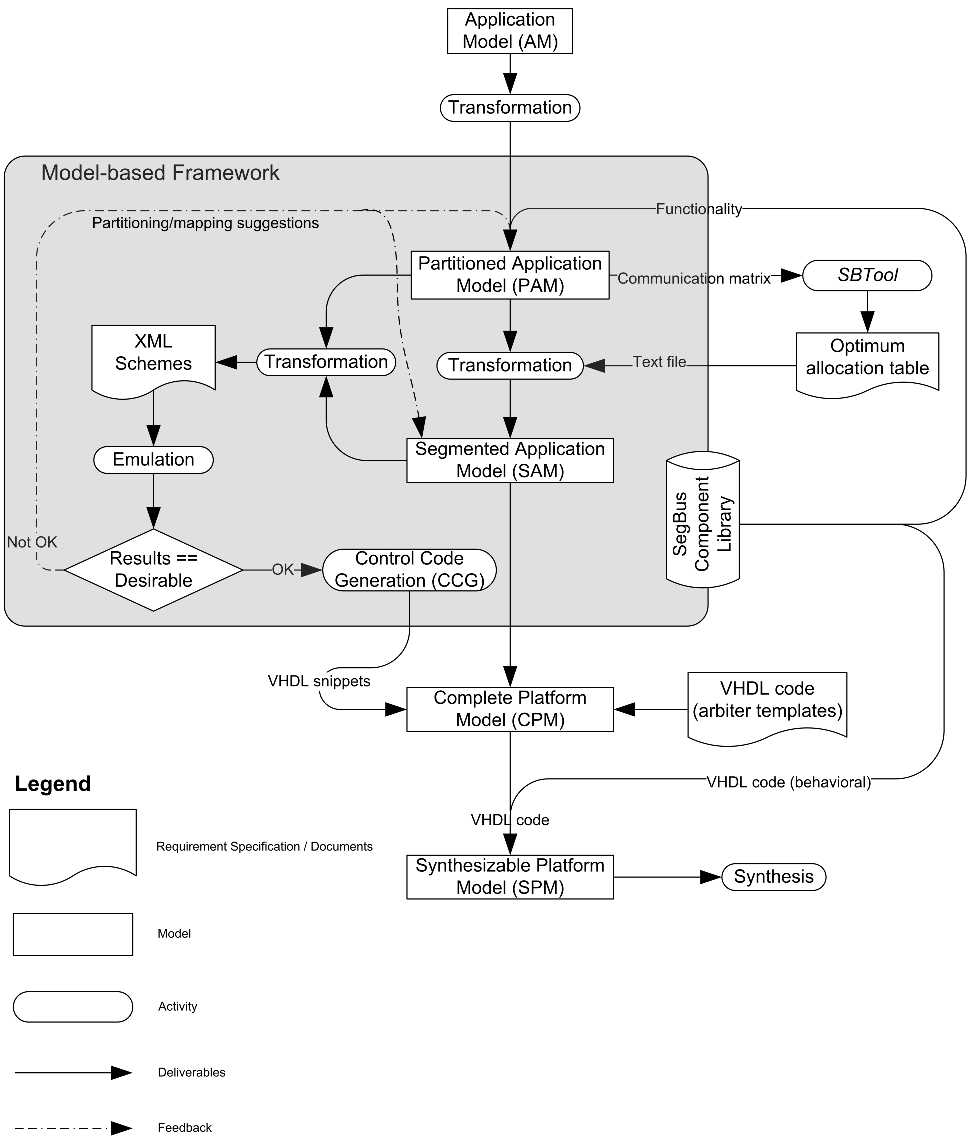


Figure 3.1: The macro-level view of the *SegBus* design process based on the proposed framework.

Taking from the designer parameters such as the number of segments, the (independent) *SBTool* [19] analyzes the given communication matrix and provides support for an initial allocation of the application functional elements onto the platform. The allocation information, stored as a text file, serves as input into the model based framework. Here, with the *SegBus* DSL support, the allocation information is then modeled to obtain the segmented application model (SAM). A DSL based approach guarantees the (structural) correctness of the SAM.

The SAM further produces data for an emulator to simulate the system, at high abstraction levels. The emulator produces two kinds of information. First, through visual representation, the performance of the SAM can be evaluated. Second, control code is obtained as VHDL snippets for the local and central arbiters. If the emulation results do not satisfy the target performance metric, the designer is able to change, within the framework, the initial allocation of modules to segments. Subsequently, after a new emulation round, the new results are available.

The VHDL code for arbiters plus the final allocation contained by the SAM converge into a complete platform model (CPM). From here, with the addition of the VHDL code for platform elements such as functional units, specific platform units (border FIFOs, templates for arbiters, synchronizers, etc), a synthesizable platform model (SPM) is obtained.

The generalized term for SAM, CPM and SPM is the platform-specific model (PSM) because of their strict dependence on the communications platform. This thesis thus uses the generalized term in the rest of the chapters to eliminate ambiguity between different models.

3.2 Model-Based Development Paradigm

As discussed in Chapter 1.2, the challenging task of system development with ever increasing complex requirements can be tackled by increasing abstraction levels, and therefore model-driven paradigm has already been proposed some time ago. In this paradigm, the dominant entities are the *visual/graphical models* which have shifted the traditional paradigm of system specification from *writing* source code to *modeling* visual models.

By observing the trends in design paradigm evolution, the *Model Driven Architecture* (MDA) [61] was launched by the Object Management Group (OMG). MDA provides a set of guidelines which designers use to specify the system requirements in a structured manner. The resulting system specifications are expressed as models. MDA supports modeling of a system from the very beginning to an implementation. In the early stage of the modeling process, MDA initially focuses on modeling the system behavior without considering a specific implementation technology. When the initial system model is developed, the modeling process gradually moves toward mapping the system model onto a specific implementation technology. This is done by *refining* and *transforming* the initial system model iteratively until it is completely mapped.

To provide a classification among different models, MDA specifies three distinct *abstraction levels* (also called *viewpoints*) for the system-under-development: a *computation independent* model (CIM), a *platform independent* model (PIM) and a *platform specific* model (PSM) [61].

The CIM targets the *context* and requirements of the system. It does not show its internal structure and rather focuses on the context where the system will be used. The PIM focuses on the functionality of the system without considering the implementation technology or platform. Finally the PSM combines the PIM with additional focus on the implementation platform to be used by the system. The *model transformation* is the key feature of MDA which transforms the system specification from one abstraction level to another. There are four different types of transformations that can be applied at certain abstraction levels, as listed below.

1. *PIM-to-PIM transformation* is applied on a PIM to get a *refined* PIM. Here, refinement corresponds to adding more details in existing model(s).

2. *PIM-to-PSM transformation* incorporates "implementation platform" details in the platform-independent PIM specification of a system.

3. *PSM-to-PSM transformation* is purposely used for refining the PSM.

4. *PSM-to-PIM transformation* is used to remove platform/technology details from a PSM, to get a PIM model.

The MDA approach is supported by a number of different technology standards which are also proposed by OMG [58] as listed below.

- Unified Modeling Language [65] to visualize, specify, construct and document software systems.

- Meta-Object Facility (MOF) [66] to define new modeling languages.

- XML Metadata Interchange (XMI) [67] to interchange and integrate models.

- Common Warehouse Metamodel (CWM) [68] to interchange warehouse and business intelligence "metadata".

- Object Constraint Language (OCL) [64] to get "consistent" models.

In the following, two MDA's technology standards, that is, UML and OCL are discussed in section 3.3 and 3.4 respectively. Next, the *SegBus* UML profile is discussed in section 3.5. The application modeling using the P-SDF is described in section 3.6. Furthermore, the *SegBus* DSL, the *SegBus* emulator and the VHDL snippets are briefly discussed in section 3.7, 3.8 and 3.9 respectively.

3.3 The Unified Modeling Language

As discussed in Chapter 1.2, when the software engineering community realized a need to increase the abstraction levels of developing systems, the *object-orientation* emerged into practices. The community appreciated during that time, and still appreciates its powerful features like data encapsulation, inheritance, polymorphism, etc., to help *classify* different involved *objects*.

With the gradual adoption of the object-oriented approach in different technical areas, many new programming and modeling languages, and methods have been developed to support the approach, each with its strengths and weaknesses. Hence, few of them were adopted by the community on a significant scale, whereas remaining were either combined with other methods or simply filtered out. The three major methods of that time, Booch methodology [10], the Object Modeling Technique (OMT) [11] and Object-Oriented Software Engineering [12], later started to show similarities with the other methods and their promoters, most importantly Grady Booch, James Rumbaugh and Ivar Jacobson, then joined hands to *unify* common concepts of object-orientation. Consequently, the *Unified Modeling Language* (UML) has been introduced in 1996, as UML 0.9 and UML 0.91 afterward, and made a prominent successor position among other similar languages in the object-oriented wave. In 1997, by a standardization process launched by OMG [58], it became one of the OMG standard as UML 1.0. Afterward, many new versions have been introduced by the time with improvements and additional features. Here, this thesis uses UML version 2.2 [65], since it was the latest version at the starting time of this research work. However, the approaches discussed in this thesis may be applied with the latest versions of UML, too. In addition, the backward compatibility of the proposed approach has not been examined with the older versions of UML.

The modeling concepts of UML are grouped into 14 *language units* [65] - see Figure 3.2. A language unit or UML diagram consists of a set of tightly-coupled modeling concepts that enable users to model a certain aspect of the system-under-modeling. A language unit is also known as *UML diagram*. All the language units are further classified into two distinct classes, namely as, *structural UML diagrams* and *behavioral UML diagrams*.

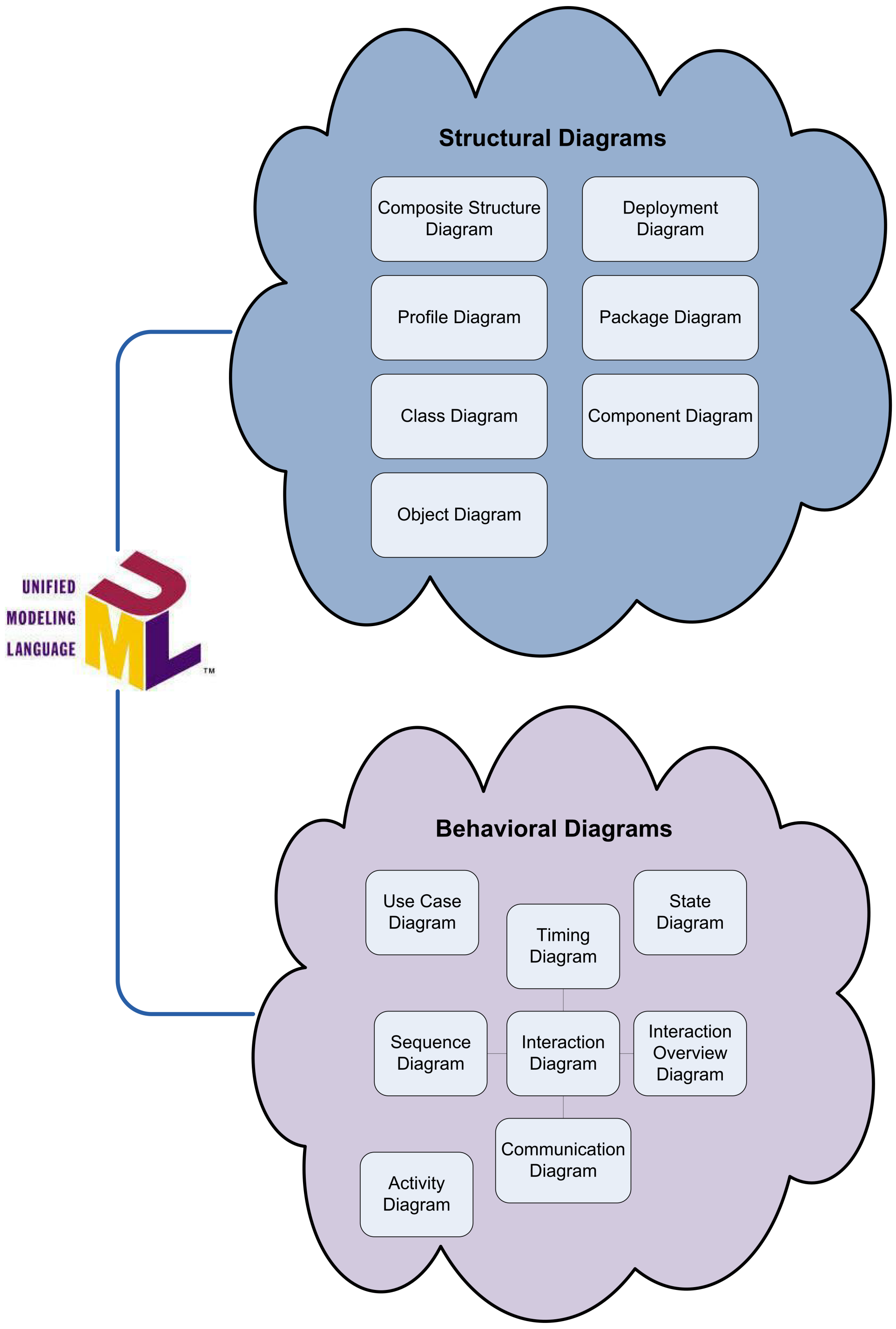


Figure 3.2: UML language units / diagrams.

The structural UML diagrams consist of constructs which are used to define structure of the system or sub-systems at a particular level of abstraction. For instance, the *Class diagram* provides a *static* structure by modeling the system in terms of data types (classes), their properties (attributes, operations) and relationships among them (e.g., associations, containment, inheritance, etc.). The *Object diagrams* are used to represent the structural relationships between classes' instances (i.e., *objects*). The *Package diagrams* show packages of classes and the dependencies among them. Each package contains a set of classes that have similarities in some aspect(s).

On the other hand, the behavioral UML diagrams consist of constructs which are used to model the functionality and operations of the system. For instance, the *Use Case diagrams* enable the capture system requirements by modeling the relationships between *actors* (external user of the system) and *use cases* (system functionality). The *State diagram* provides a view of the behavior of objects, by modeling all possible *states* of each object and *transitions* between these states. The *Activity diagrams* models the sequencing of activities performed by the system or sub-system.

Generally, not all the 14 diagrams need to be employed at once for any system modeling. Instead, the designer has to select the ones needed for a given aspect of the system model. UML is a "general-purpose" modeling language which limits the degree of its usage in every application domain. Therefore, to increase its suitability for every particular application domain, OMG specifies two important techniques to introduce domain specific languages for the concerned domains. The first technique is to define a new language (alternative to UML) applying the same principles as for UML definition. The second method is the UML language's extensibility support. In brief, the UML language supports extensibility mechanisms where the designer can customize the existing UML constructs by defining *profiles* as per needs. A *UML profile* is a specialized package consisting of a set of *stereotyped classes* (defined below). This is done by extending the elements of UML language to represent entities of a particular *application domain.* This extension process leads to a new *UML profile* with the help of three elements: *Stereotypes*, *Tagged values*, and *Constraints.* By stereotyping UML elements, the designer *specializes* a UML element for a particular concept or entity in the application domain. With tagged values, the designer defines certain attributes of these stereotyped elements. And with constraints, the designer enforces certain additional restrictions of usage on the stereotyped elements.

3.4 The Object Constraint Language

The Object Constraint Language (OCL) [64] enables the writing of *expressions* that specify certain constraints for different UML constructs and it is now part of the UML standard since UML version 1.3. The constraints written in OCL are applied on a given UML model to make the model precise, complete and consistent. Initially, UML was not expressive enough to describe each and every aspect of the system under modeling, and natural language was used as a replacement to describe certain aspects of the system which eventually led to ambiguities. Later, formal languages were introduced to overcome this problem. However, they were quite mathematical in nature, and hence, not easy to use. Therefore, OCL was introduced to deal with all such problems.

OCL is not a general-purpose programming language, it does not support control flow statements as other general-purpose programming languages do. Using OCL, constraints can be defined in the form of *invariants*, *precondition*, *postcondition*, *guard*, etc. The invariants are Boolean OCL constraints which evaluate to true/false and are attached to classes, while precondition and postcondition constraints are attached to operation or methods which are executed pre- and post execution of the relative operation/method respectively. OCL's guard expressions are applied on UML's "state transition diagram" and the guard expressions specify conditions which need to be met before a transition can occur.

Since, OCL is a pure specification language [64], an OCL expression does not introduce any kind of side effects on the target model and simply returns a value. Moreover, the evaluation of an OCL expression does not cause the state of the system to change. However, a specific state change, for example, in a postcondition of an operation/method, can be specified by an OCL expression without any potential risk on the system state. In summary, a model can be made more expressive and complete by adding constraints, and OCL provides a formal and non-ambiguous way of achieving this goal.

3.5 The SegBus UML Profile

Lindroth et. al [38] discussed the initial steps of the *SegBus* platform profile. It includes a hierarchical decomposition of platform components and provides appropriate means for characterization, instantiation and connectivity. However, some important features were missing, such as model validation according to platform definition, attributes of platform elements, structural constraints, etc. Figure 3.3 shows the profile elements.

The profile contains the structural elements of the platform. It contains the platform itself, the stereotype *SegBusPlatform*, one element modeling the segments, *Segment*, the stereotype representing the ***SA***, *SegmentArbiter*,

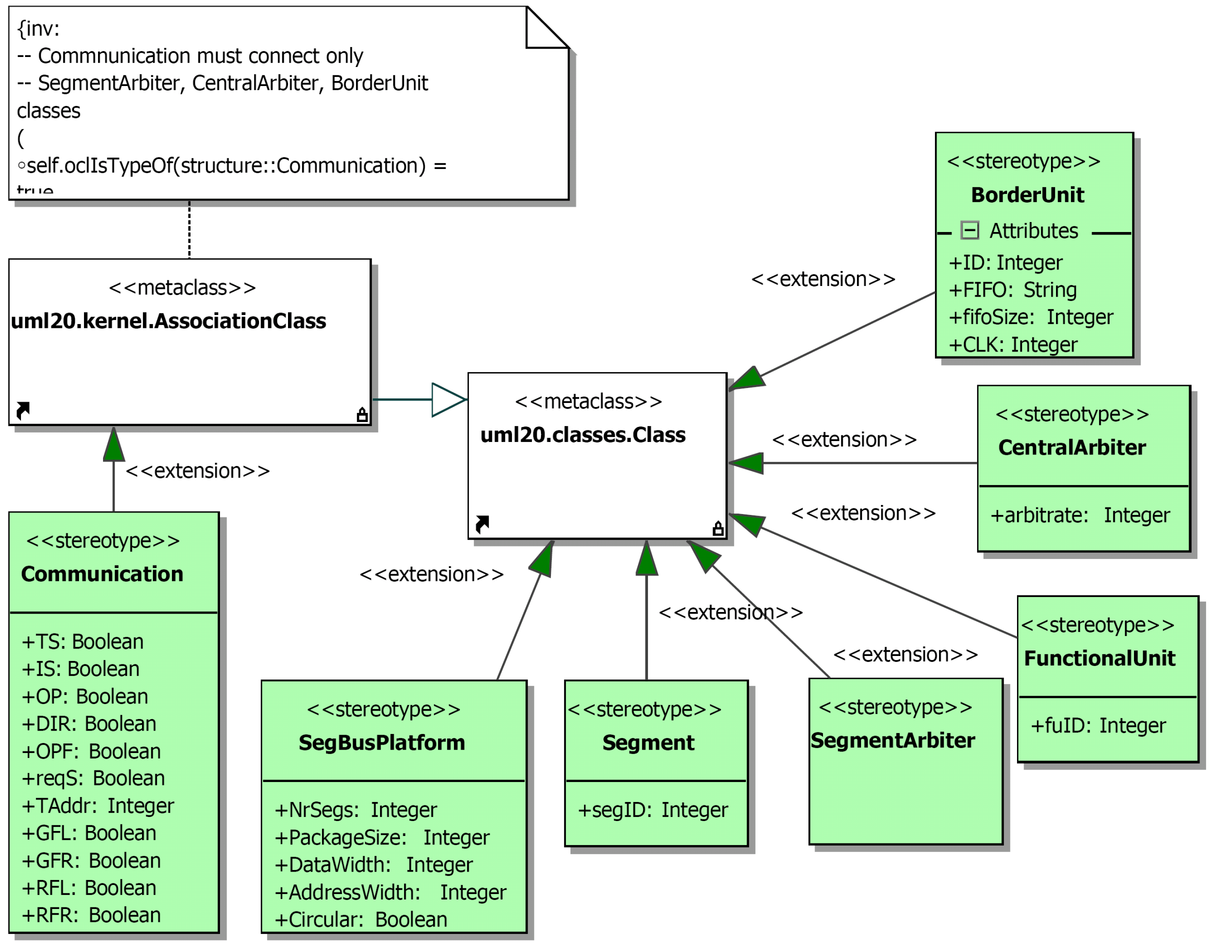


Figure 3.3: The SegBus profile elements [38].

stereotype *CentralArbiter* represents ***CA***, etc. A *metaclass* is a class whose instances are classes. Here, all the elements are generalization of metaclass *uml20.classes.Class*. Based on the ideas presented in [38], this thesis builds a similar and enhanced profile for the *SegBus* platform to be able to model systems in a domain-specific manner within a feature-rich framework.

3.6 Application Modeling using the P-SDF

The specification of the application enters the model-based framework as a platform-independent *Package SDF* (P-SDF) model [III][IV]. This means that the application is already partitioned on functional elements available from the library. P-SDF is a customized version of Synchronous Data Flow diagrams [37]. The approach is intended to facilitate the mapping of the application to the architecture due to the similarity between the operational semantics of the P-SDF and that of the *SegBus* architecture, thus allowing us to cope in a more detailed manner with the communication characteristics of the *SegBus* platform.

"An *architectural style* determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of *constraints* on how they can be combined" [6]. The P-SDF is a specialization of the *Pipes and Filters* architectural style (See [6] for description of this style of software architecture).

A P-SDF comprises mainly two elements: *processes* and *data flows*; data is organized in packages according to package size during execution. Processes transform input data packages into output ones, whereas package flows carry data from one process to another. A *transaction* represents the sending of one data package by one source process to another target process, or towards the system output. A *package flow* is a tuple of four values, P_t, D, T and C. The P_t value represents the target process for the given transactions; the D value represents the number of data items emitted by the same source towards the same destination successively; the T value is a relative ordering number among the (package) flows in one given system; and the C value represents the number of clock ticks a process consumed before sending one package. Thus, a flow is understood as the number of data items (later transformed into packages issued by the same process, targeting the same destination, having the same ordering number and same clock ticks require to process one individual package.

If s is the package size (number of data items in a package) in the platform configuration, then the *Package SDF (P-SDF)* of a certain system is a sequence of package flows, $< (P_{t_x}, \frac{D_1}{s}, T_1, C_1), \ldots, (P_{t_x}, \frac{D_n}{s}, T_n, C_n) >$, where $\forall i, j, x \in \{1, \ldots, n\} \cdot \frac{D_i}{s} \neq \frac{D_j}{s}$ and $T_1 \leq T_2 \leq \ldots \leq T_n$.

The above definition of the P-SDF establishes a possibility of several flows to exist on a very same T value thereby a deep level of parallel execution can be set up, if possible as per application requirements [50][36][III].

3.7 DSL for the SegBus Platform

Domain-specific modeling (DSM) is a way of designing systems that involves the systematic use of domain-specific languages (DSLs) to represent the various facets of a system. A DSL tend to provide higher-level abstractions than general-purpose modeling languages like UML. It encapsulates domain concepts and provide semantics to domain entities, allowing designers to work directly with domain concepts. Here, the proposed research employs the *MagicDraw UML* [60] tool to graphically model various artifacts of the proposed DSL, as the tool not only provides UML-based capabilities, but it also provides the *DSL Customization Engine* - a mechanism able to process user-defined rules for the DSL, and provide a visual interface for validation.

Figure 3.4 shows a more detailed and micro-level view of the proposed design framework which highlights its certain primary features and internally involved design stages. Within the framework, the thesis transforms the top level platform concepts into the high-level graphical constructs to form a DSL, specific for the *SegBus* platform. The DSL provides a graphical environment where a designer can model P-SDF and PSM models of the application quickly and assign pre-existing components from the *SegBus Component Library* during the modeling. The application should be already partitioned before modeling it in the P-SDF form and mapping it on to the platform according to available library components.

The model can be validated for possible mistakes to get the correct P-SDF and PSM. Later on, the designer transforms P-SDF and PSM models of the application into XML schemes using M2T transformation supplied by the tool. The XML schemes contain information about platform elements, application components and their relative placement on different segments.

The thesis developed the "*SegBus* DSL" over three main directions: *Profile Development*, *DSL Customization*, *Structural Constraints*. A detailed analysis of the DSL development is discussed in Chapter 4.

3.7.1 The SegBus Component Library

The *SegBus* components' library consists of a number of *record locators*. A record locator stores important information about the library component. The information includes a pointer to actual location of a VHDL file that implements the functionality, the name, the version, and technical specifications, such as area of the component in the given technology, timing

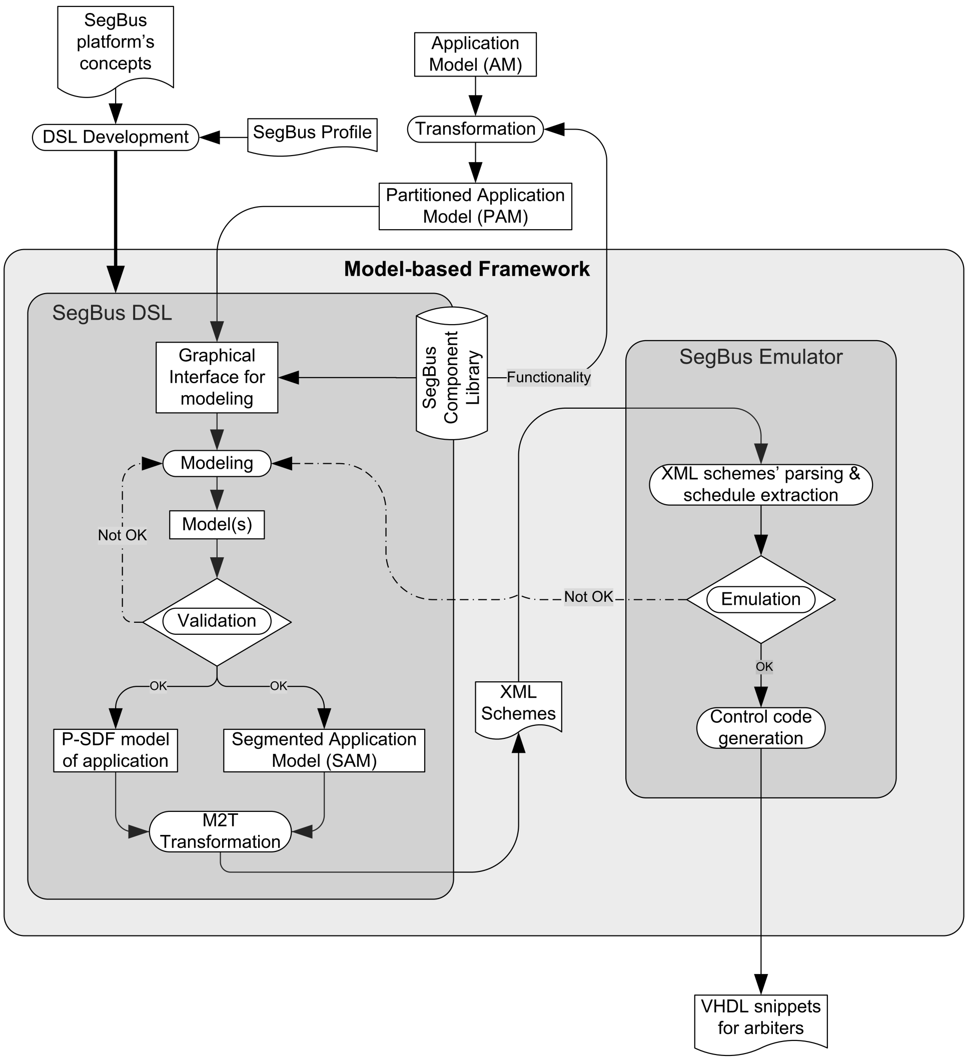


Figure 3.4: The micro-level view of the proposed framework and its internal distinct units.

to process the inputs (clock ticks), etc. Figure 3.5 depicts the idea more conveniently.

The framework utilizes the library especially while modeling the PAM and SPM. During the PAM development phase, the designer refers the library to ensure about the availability and variety of functional components and partitions the application accordingly. Further, while developing the SPM, the library used more thoroughly for pointing out the actual VHDL implementation of the selected components and to increase the visual understanding of the system. A detailed description about the development of the library is presented in Chapter 4.4.

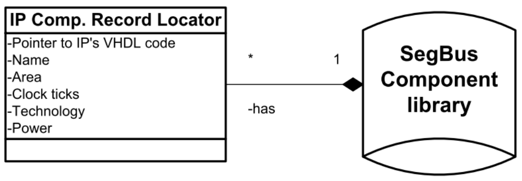


Figure 3.5: The SegBus library component structure.

3.7.2 Extensible Markup Language

A *markup* text is special text which can be distinguished by other texts *syntactically*. The markup texts are used to give special meanings (annotation, organization, ..) to ordinary text documents. *Extensible Markup Language* (XML) is a markup language consisting of a set of rules that are used to format data and information in *structured documents*. These XML documents are easily understandable by both humans and machines. Moreover, the data represented in XML documents are also *machine-processable*, which eventually, opens doors for information transportation between different types of applications.

XML was developed to structurally represent and share different types of data between a variety of domains. This happened when the computing shifted from mainframes to client-server model. The hassle of information communication, representation and understanding of the received information between local and remote machines had shifted the focus to markup languages [14]. On one hand, the challenges of information containment and visualization between computer applications and human had been resolved by the *HyperText Markup Language* (HTML). On the other hand, the challenges pertaining to information representation and transportation between different computer applications (no matter where they are located) has been tackled with the advent of XML. For instance, in one application domain, the data from the domain entities (e.g., applications' objects) are organized into XML document(s), and then transfer to another application domain. The entities at the receiving domain upon receiving the XML document transform the XML document(s) into a form which the receiving domain generally uses inside its boundary.

XML documents can be made in any suitable desired format so that information can be stored, represented and transported. However, problems may occur with respect to formatting of the documents, if they are to be shared with others. To enforce certain protocol in the formatting process, the designers use *XML schema* to formalize protocol for formatting the XML documents which to be shared among different users/applications in a trouble-free manner. In this case, each XML document have to conform with particular XML schema/schemes.

In the proposed framework, the designer transforms application and platform models into XML schemes, such that the models can be shared with emulator application, which eventually reads and processes the models (in the form of XML schemes) for its own purposes.

3.7.3 Model Transformation

The *model transformation* is an important tool of the model-driven paradigm which transforms a set of *source models* from one abstraction level into a set of *target models* to the same or to different abstraction level(s). If the transformation takes place between the same abstraction level, it is called *horizontal transformation* e.g., PIM-to-PIM and PSM-to-PSM transformations; and if it takes place between two different abstraction levels, then it is called *vertical transformation* e.g., PIM-to-PSM transformation, etc.. Both source and target models must have to *conform* respective (same or different) metamodel(s). This is done by defining how elements in the source models should appear in the target models by relating respective elements in the source and target metamodels. In model-driven paradigm, it is mainly used to refine and evolve the abstract specifications into more concrete ones in a step-by-step manner.

There are two main kinds of *model* transformation: *Model-to-Model* (M2M) and *Model-to-Text* (M2T). In M2M transformation, the transformation process transforms one set of different models into another set of models and hence, it is called model-to-model transformation. For example, the transformation of one set of UML class diagrams into another *refined* set of class diagrams, etc.; While in M2T transformation, a set of models are transformed into a set of text-based elements. For example, a transformation of UML models into source code of some programming language, XML documents/schemes, etc.

The proposed framework employs Model-to-Text (M2T) transformation for transforming the system models, which are built using the DSL, into

XML schemes to be further used by *SegBus Emulator* (discussed below) for examining the merits of the source models for possible implementation.

3.8 The SegBus Emulator

Generally, emulation is necessary while designing applications targeting hardware devices and platforms. The huge design and manufacturing costs of such hardware platforms motivate designers to develop emulators and verify the execution results. An emulator provides, to a certain extent of accuracy, the same functionality as the original hardware platform or computer program. Designing an emulator requires a thorough understanding of the target device or platform. This thesis introduces the *SegBus* emulator to test platform configuration and estimate performance aspects before moving towards the final implementation.

In the proposed framework context, before the execution, the emulator application reads the XML schemes of the P-SDF and PSM models, package size and considers the structure (segment organization and resource allocation) from the XML schema of the PSM. Upon completion, the tool returns results of the transactions from each platform element, performed during execution. At this stage, if the obtained performance aspects of the emulated configuration is up to an optimum level, the designer generates the control code for the arbiters.

The details on the emulator are described in Chapter 5.

3.9 The VHDL Snippets

The control of the communication scheduling is done with the decision and the supervision of the local and central arbiters. Templates for the operation of these modules are located in the library. Based on the input from the emulator, these are updated with the schedules for operation, trying, at the same time, to enable an as much as possible parallel perspective on the data transports and processing.

The details on how the snippets are defined and obtained are described in Chapter 6.

3.10 Summary

This chapter presented a general overview of the proposed model-based development and verification framework together with a brief description of the employed methodology. The methodology can be seen as being inspired by the Y-chart approach. Hence, the methodology has to be followed while designing systems using the proposed framework. This chapter also introduced the important cornerstones that guide us towards building the framework successfully. In the following chapters, the thesis will discuss in more detail the important cornerstones of the framework, that are required in different phases of the methodology.

Chapter 4

SegBus DSL

This chapter presents a *Domain Specific Language* (DSL) for the *SegBus* platform. The DSL, based on a UML profile, consists of graphical platform elements in the form of stereotypes with the necessary tagged values to depict platform aspects at high level of abstraction. Customizations are applied to each stereotyped element in the form of user-defined rules to restrict relationship between platform elements. The *Object Constraint Language* (OCL) is employed to introduce constraints, in order to impose structural requirements between platform elements, for which we introduce mechanisms to validate them. Moreover, a brief description about the *SegBus* components library and its usage is also presented. Figure 4.1 shows the high-level design entry of any target application into the proposed framework where it

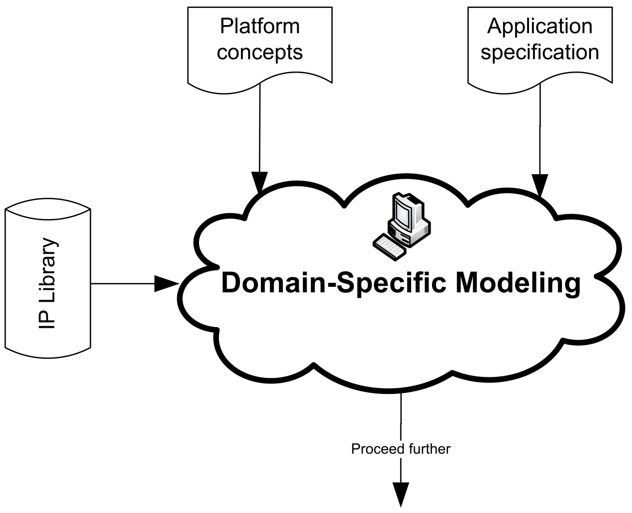


Figure 4.1: High-level design entry to application development.

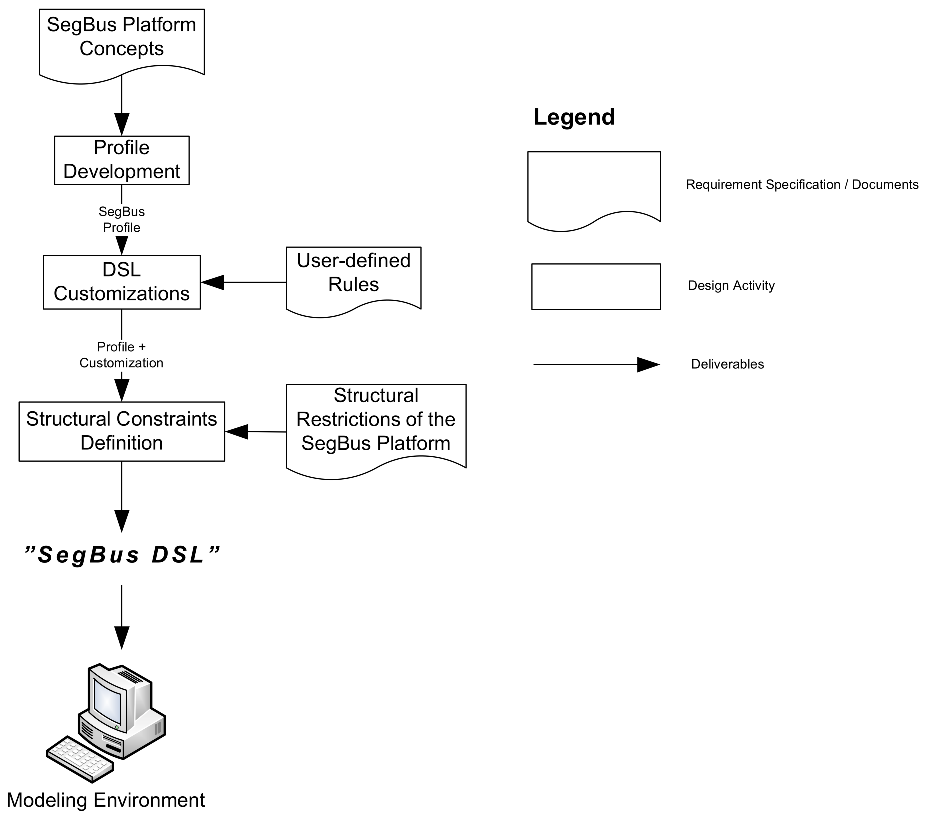


Figure 4.2: Design activities involved in the development of the *SegBus* DSL.

is modeled in order to be implemented on the *SegBus* platform at the later stages of the design methodology.

4.1 Profile Development

The structural characteristics of the platform are the key starting point of profile development. These are analyzed in order to select the UML elements for modeling the hardware components of the platform at high levels of abstraction. The important consideration in this phase is the selection of suitable UML elements which can preserve related semantics and graphically represent the platform elements in the application domain. The *UML class diagram* describes the structural view of the system. Thus a UML package - *SegBusProfileMagic* is built which is a collection of classes in a class diagram with necessary stereotypes, in order to sustain application development on the *SegBus* platform. The profile defines the main structural elements of the platform. All the classes in the profile that model

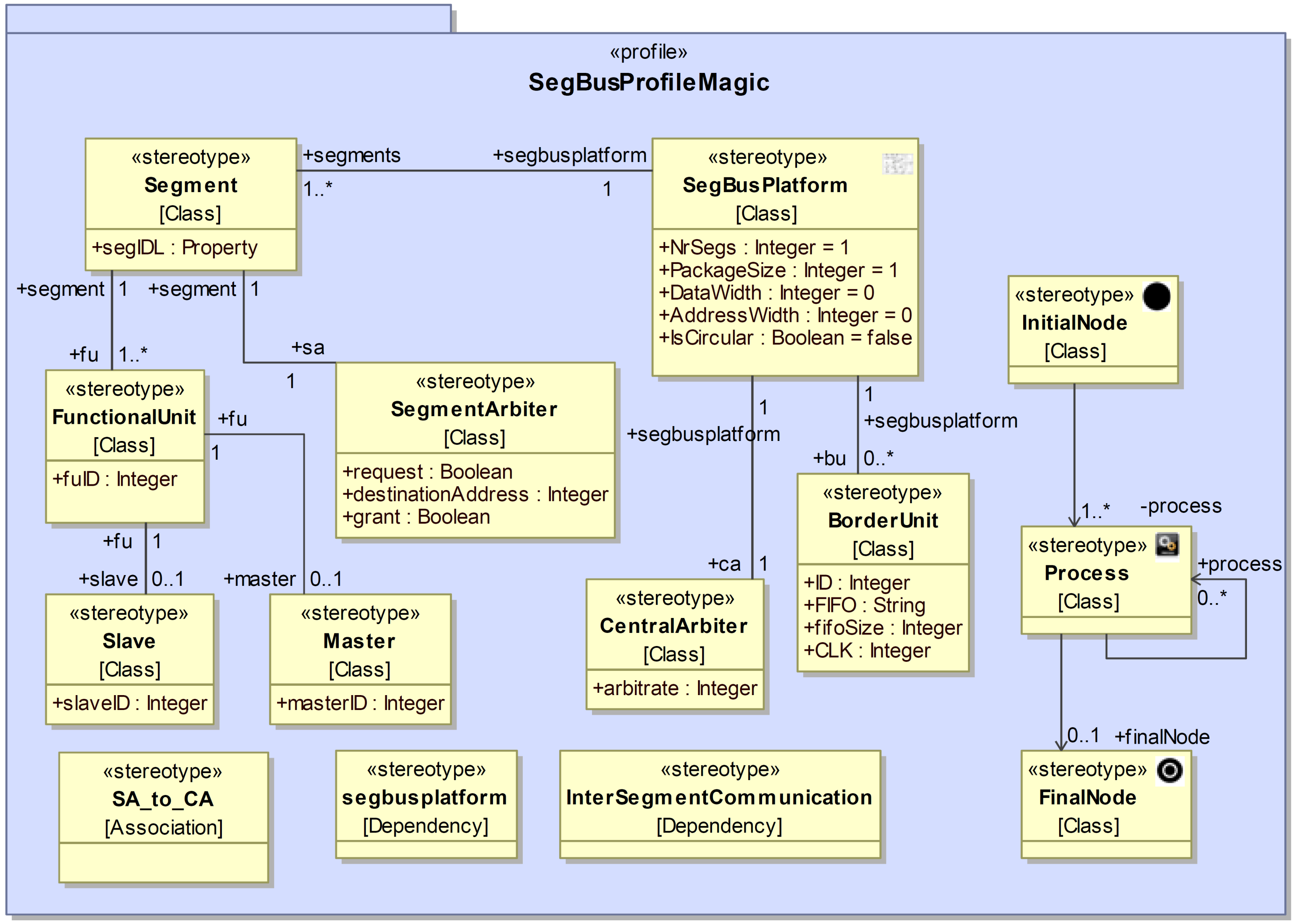

«profile»
SegBusProfileMagic
«stereotype» Segment [Class]
+segIDL : Property
+segments
1..*
+segbusplatform
1
«stereotype» SegBusPlatform [Class]
+NrSegs : Integer = 1
+PackageSize : Integer = 1
+DataWidth : Integer = 0
+AddressWidth : Integer = 0
+IsCircular : Boolean = false
+segment
1
+fu
1..*
+segment
1
+sa
1
«stereotype» FunctionalUnit [Class]
+fuID : Integer
+fu
1
«stereotype» SegmentArbiter [Class]
+request : Boolean
+destinationAddress : Integer
+grant : Boolean
+segbusplatform
1
+ca
1
1
+segbusplatform
+bu
0..*
«stereotype» InitialNode [Class]
+fu
1
+slave
0..1
+master
0..1
«stereotype» Slave [Class]
+slaveID : Integer
«stereotype» Master [Class]
+masterID : Integer
«stereotype» CentralArbiter [Class]
+arbitrate : Integer
«stereotype» BorderUnit [Class]
+ID : Integer
+FIFO : String
+fifoSize : Integer
+CLK : Integer
1..*
-process
«stereotype» Process [Class]
+process
0..*
0..1
+finalNode
«stereotype» FinalNode [Class]
«stereotype» SA_to_CA [Association]
«stereotype» segbusplatform [Dependency]
«stereotype» InterSegmentCommunication [Dependency]

Figure 4.3: Platform elements and their association in profile.

a particular element of platform are generalizations of the metaclass *UML Standard Profile::UML2 Metamodel::Classes::Kernel::Class*. The structural view of the profile is depicted in Figure 4.3 with the necessary association and multiplicities between profile elements. Figure 4.2 provides a high-level description of the activities involved in the development of the *SegBus DSL*.

The platform (*SegBusPlatform*) is characterized by the number of segments it contains, platform geometry (linear/circular), package size for communication, data width and address lines. The *BorderUnit* element is an interface between one segment and its neighbors. The internal FIFO buffer is characterized by the *fifoSize* tag. The ***FU***'s ID (natural number that is unique at system-level) is inherited by both contained *Master* and *Slave*. The ***FU*** methods contain procedures to produce data and to communicate with the same segment's ***SA***. Procedures for sending and receiving data are placed within *Master* and *Slave* respectively.

Additionally, Figure 4.3 contains three more stereotyped elements to model the application at P-SDF level, that is, *InitialNode*, *Process* and *FinalNode*. The initial node can only be connected to one or more process node(s). The process node can be connected with other process nodes and/or to the final node.

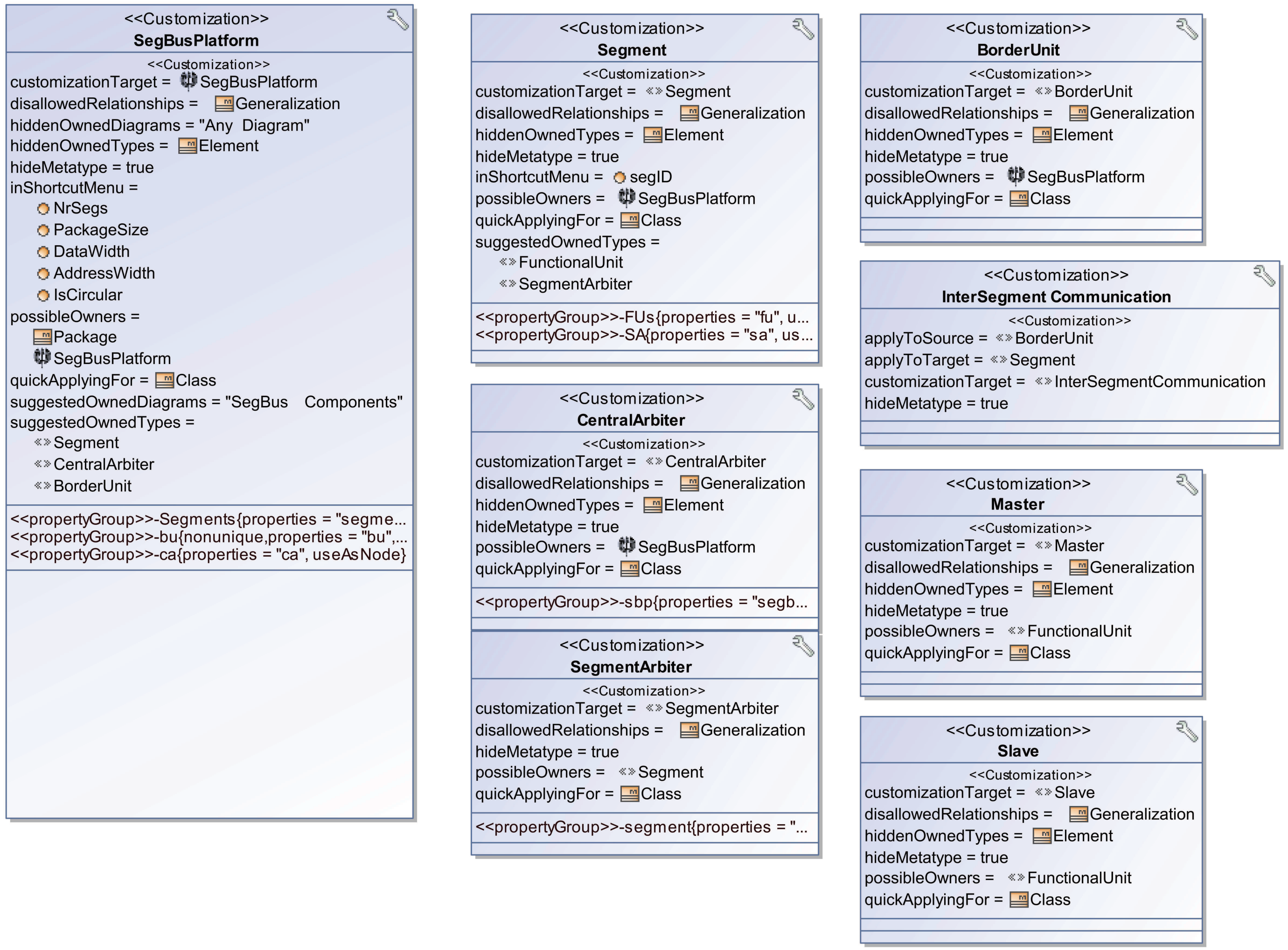
4.2 DSL Customizations

The next step in DSL development is to introduce user-defined rules for each profile element so that the modeling tool [60] fully adapt the proposed DSL in its modeling environment. All user-defined rules for each profile element are stored in customization classes. Customization classes are generalization of metaclass *UML Standard Profile::MagicDraw Profile::DSL Customization::Customization* class, with stereotype *Customization*. These customization classes consist of tags that store the user-defined DSL customization rules. The customization rules are parsed and interpreted by the *DSL Customization Engine* to assist validation process. A UML package is created to store all customization classes.

Figure 4.5 shows the user-defined rules for each profile element. The usage of a few customization rules are described below.

- **customizationTarget.** This tag stores the names of stereotype(s) which are going to be customized with respect to current class. User-defined rules that are introduced in the customization class will be applied to all stereotyped classes which are mentioned in this tag (first column in Figure 4.5).

- **possibleOwners.** This tag consists of stereotyped or other UML elements that can instantiate current element. The *possibleOwner* of



```
<<Customization>>
SegBusPlatform
<<Customization>>
customizationTarget = SegBusPlatform
disallowedRelationships = Generalization
hiddenOwnedDiagrams = "Any Diagram"
hiddenOwnedTypes = Element
hideMetatype = true
inShortcutMenu =
    NrSegs
    PackageSize
    DataWidth
    AddressWidth
    IsCircular
possibleOwners =
    Package
    SegBusPlatform
quickApplyingFor = Class
suggestedOwnedDiagrams = "SegBus Components"
suggestedOwnedTypes =
    Segment
    CentralArbiter
    BorderUnit
<<propertyGroup>>-Segments{properties = "segme...
<<propertyGroup>>-bu{nonunique,properties = "bu",...
<<propertyGroup>>-ca{properties = "ca", useAsNode}
```

```
<<Customization>>
Segment
<<Customization>>
customizationTarget = Segment
disallowedRelationships = Generalization
hiddenOwnedTypes = Element
hideMetatype = true
inShortcutMenu = segID
possibleOwners = SegBusPlatform
quickApplyingFor = Class
suggestedOwnedTypes =
    FunctionalUnit
    SegmentArbiter
<<propertyGroup>>-FUs{properties = "fu", u...
<<propertyGroup>>-SA{properties = "sa", us...
```

```
<<Customization>>
CentralArbiter
<<Customization>>
customizationTarget = CentralArbiter
disallowedRelationships = Generalization
hiddenOwnedTypes = Element
hideMetatype = true
possibleOwners = SegBusPlatform
quickApplyingFor = Class
<<propertyGroup>>-sbp{properties = "segb...
```

```
<<Customization>>
SegmentArbiter
<<Customization>>
customizationTarget = SegmentArbiter
disallowedRelationships = Generalization
hideMetatype = true
possibleOwners = Segment
quickApplyingFor = Class
<<propertyGroup>>-segment{properties = "...
```

```
<<Customization>>
BorderUnit
<<Customization>>
customizationTarget = BorderUnit
disallowedRelationships = Generalization
hiddenOwnedTypes = Element
hideMetatype = true
possibleOwners = SegBusPlatform
quickApplyingFor = Class
```

```
<<Customization>>
InterSegment Communication
<<Customization>>
applyToSource = BorderUnit
applyToTarget = Segment
customizationTarget = InterSegmentCommunication
hideMetatype = true
```

```
<<Customization>>
Master
<<Customization>>
customizationTarget = Master
disallowedRelationships = Generalization
hiddenOwnedTypes = Element
hideMetatype = true
possibleOwners = FunctionalUnit
quickApplyingFor = Class
```

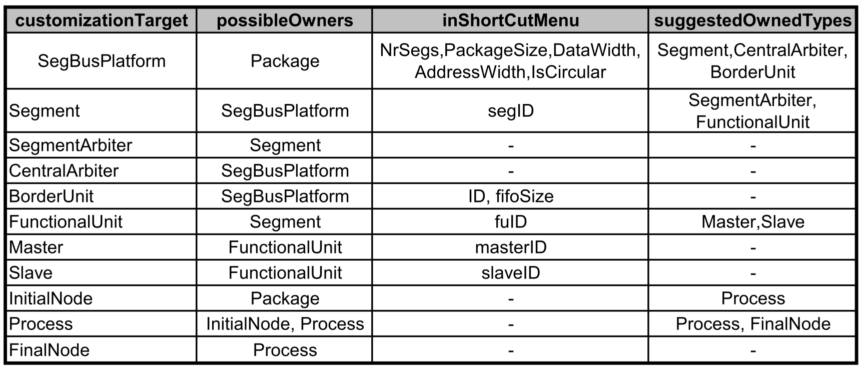
```
<<Customization>>
Slave
<<Customization>>
customizationTarget = Slave
disallowedRelationships = Generalization
hiddenOwnedTypes = Element
hideMetatype = true
possibleOwners = FunctionalUnit
quickApplyingFor = Class
```

Figure 4.4: DSL Customization classes for each element of the *SegBus* platform.

Segment can only be the *SegBusPlatform* (second row of Figure 4.5), which can instantiate it.

- **inShortCutMenu.** This tag is used to add attributes of a class in shortcut menu. In the first row of Figure 4.5, the properties that appear as shortcut menu items are included in the context of the *SegBusPlatform* class.

- **suggestedOwnedTypes.** This tag contains list of stereotypes and metaclasses whose object can be instantiated inside the stereotyped class as inner elements. For instance, *SegBusPlatform* can only be associated with *Segment, CentralArbiter* and *BorderUnit* stereotyped classes.



customizationTarget	possibleOwners	inShortCutMenu	suggestedOwnedTypes
SegBusPlatform	Package	NrSegs,PackageSize,DataWidth, AddressWidth,IsCircular	Segment,CentralArbiter, BorderUnit
Segment	SegBusPlatform	segID	SegmentArbiter, FunctionalUnit
SegmentArbiter	Segment	-	-
CentralArbiter	SegBusPlatform	-	-
BorderUnit	SegBusPlatform	ID, fifoSize	-
FunctionalUnit	Segment	fuID	Master,Slave
Master	FunctionalUnit	masterID	-
Slave	FunctionalUnit	slaveID	-
InitialNode	Package	-	Process
Process	InitialNode, Process	-	Process, FinalNode
FinalNode	Process	-	-

Figure 4.5: User-defined rules for different attributes of the Customization classes.

We've also introduced three different customized *Dependency* links (Figure 4.3), in order to connect different stereotyped elements of the *SegBus* platform according to needs. The advantage of customizing such links during DSL development is to specify what will be the possible source and target stereotype(s) for given links. The customization of these links allows designer to connect only particular platform elements by imposing user-defined rules. In Figure 4.6, the two elements *Segment* and *BorderUnit* are connected with a customized link *InterSegmentCommunication.* The link imposes specific properties of the platform for communication between mentioned platform elements.

Figure 4.4 depicts the customization classes of the *SegBus* DSL.

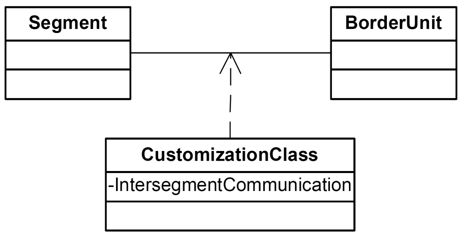


Figure 4.6: Dependency link between two profile elements.

4.3 Structural Constraints

The stereotypes defined in the *SegBus* Profile, as discussed in section 4.1, are simply a set of customized UML elements, each representing a specific element of the *SegBus* platform. In fact, these stereotypes do not enforce the structural restrictions as defined by the platform itself. The platform characteristics defined in section 2.3 need to be introduced in the form of structural constraints in the DSL. The required constraints are specified by using the *Object Constraint Language* (OCL v2.0) [64] and relate them to the *SegBus* UML profile, such that a correct component approach to platform design can be implemented. Some of the constraints are already introduced when applying multiplicities to relationships between elements in profile the development phase. For instance, there should be exactly one *CentralArbiter* in whole *SegBusPlatform* - modeled by specifying multiplicity as '1' (Figure 4.3). Also, it is important to enforce in design that the number of *Segment*s should be equal to the property *NrSegs* of *SegBusPlatform*, number of *BorderUnit*s should match the platform geometry, etc.

All the constraints are stereotyped as *validationRule* from the *Validation Profile*, a profile supplied by the tool for supporting the validation of models. A *validation suite* defines a set of validation rules, to be applied when validating a model. The purpose of making a validation suite is to group constraints logically in a UML package, *SegBus Constraints*, stereotype with *validationSuite* with proper context supplied for each constraint. The designer applies this validation suite on models when validating them against the platform constraints.

Upon any breach of any constraint requirement during the design process, the tool provides an error message with a text specified by the DSL. The designer can subsequently try to solve the indicated problem. The description of a few of the constraints that were introduced in the DSL, together with their respective error messages, are given below.

Number of Segments: This constraint enforces the number of *Segment*s in the model to be equal to the value of the integer attribute *NrSegs* that the designer specified in the stereotype *SegBusPlatform*. *NrSegs* represents the number of segments that the designer requires in the platform. The constraint specification is given as:

```
context SegBusPlatform
inv NrOfSegments:
self.segments->size() = self.NrSegs
self.segments->size() > 1
```

Error Message 1. “Number of segments in model are not same as specified in SegBusPlatform”.

Cause. The designer introduced more / less segments than the specified number.

Number of *FunctionalUnit*s in a Segment: This constraint enforces in model that each segment must contain at least one *FunctionalUnit*.

```
context Segment
inv NumberOfFU:
self.fu->size() >= 1
```

Error Message 2. “There should be at least one Functional Unit in each segment”

Cause. A segment does not contain any *FunctionalUnit*.

Number of BorderUnits: This constraint enforces in design that there must be a number of *BorderUnit*s matching to number of segments with respect to platform geometry.

```
context SegBusPlatform
inv NumberOfBorderUnits:
if self.IsCircular = true then
    self.bu->size() = self.NrSegs
else
    self.bu->size() = self.NrSegs-1
endif
```

Error Message 3. “Number of Border Units is not compliant given the selected platform topology”.

Cause. The wrong number of border units has been included in the design.

Missing *BU*s: This constraint issues an error message when the PSM model does not contain any ***BU***s whereas the segments are already existed.

```
context SegBusPlatform
inv MissingBorderUnits:
if NrSegs <> 1 then
    self.bu->notEmpty()
```

Error Message 4. “None of the Border Unit is linked with the platform instance”.

Cause. When two or more segments are already linked with the platform instance and there does not exist any ***BU***. In case when a ***BU*** is present in the model, it must not be linked with the platform instance.

Missing segment: This constraint flags an error message when none of the segments (if exist in the model) are linked with the platform instance.

```
context SegBusPlatform
inv MissingSegment:
self.segments->notEmpty()
```

Error Message 5. “None of the segments have been linked with platform instance”.

Cause. When none of the segments are linked with the platform instance (*SegBusPlatform*).

4.4 SegBus Components Library

This section discusses the development of a library of reusable IP components - *the SegBus Components Library*, which is developed and integrated into the *SegBus* DSL. Below, the thesis describes its significance in the proposed framework and its implementation approach in detail.

4.4.1 Reusability Consideration

Reusability refers to the ability of an object that can be used more than once with or without minor modification. It reduces the development time, and additionally, enables accelerated time-to-market. Here, the thesis builds a library (the *SegBus* Components Library) of often-used components to make them reusable for further developments on the platform.

Without this library, the proposed methodology lacks means of selection of the Intellectual Property (IP) components when modeling at higher levels of abstraction, and whose (selected IPs) behavior could be evaluated during emulation of a modeled system (discussed in Chapter 5). Similarly, there was no way other than to manually select and map specific IPs during the implementation phase. The thesis addresses here issues for evolving the proposed design methodology further towards reusability and versatility. The thesis introduces a library which is composed of a list of functional components to be used within the design methodology. The components are often referred during different stages of the system modeling - from high-level modeling to low-level code generation.

To achieve this research goal, thus a plug-in is built and introduced within the modeling tool [60] to get the library of reusable IP components in graphical form, which can be operated while modeling in DSL. The tool runs the plug-in every time it is run for modeling applications on the platform and provides ease of choice for components selection. The realization of *SegBus* components library is necessary because it enables to model and emulate the system more accurately due to additional information it provides such as IP name, clock ticks it consumes for processing one package, etc. This in turn makes the code generation process more straight forward and accurate.

The development of the library starts with the *use case analysis*, which is a common technique used to identify the system requirements that ultimately assists in designing classes to satisfy them. The requirements are the foundation over which the system is built on. Ambiguous and incomplete requirements lead to an incompetent system and the design efforts are thus wasted. Moreover, a *scenario* is a sequence of steps describing an interaction between a user (system designer) and a system (the *SegBus* DSL in this case) [8]. Each system requirement then evaluates to scenario(s). Further, a *use case* is a set of scenarios tied together by a common user goal.

Following important functional requirements are identified that must be included in the *SegBus* components library for a better use of it.

- The library can be opened in the tool.
- A new component can be added in the library.
- An existing component can be removed from the library.
- A component in the library can be selected and assigned to a (pre-selected) model element.
- The library can be closed.

By examining the above mentioned requirements, a use-case model is obtained, as illustrated in Figure 4.7. Following, the thesis briefly describes each of the use cases.

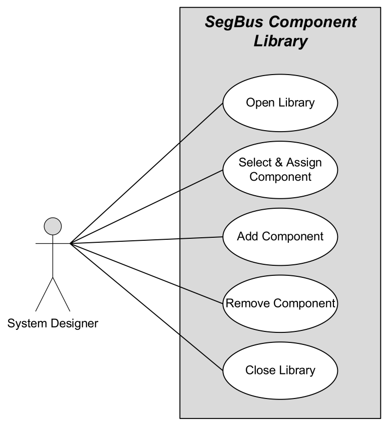


Figure 4.7: Use case model of the *SegBus* components library.

Use case 1: *Open library*

Actor	System designer
Precondition	The tool is already opened and a target model exists and contains elements to get assigned library components.
Postcondition	The library window has been opened within the modeling tool and ready to be used.
Main path (M)	1. The designer clicks "Library" menu. 2. A sub-menu "SegBus IP Library" displays. 3. The user clicks the sub-menu. 4. The library GUI displays.

Use case 2: *Close the library GUI*

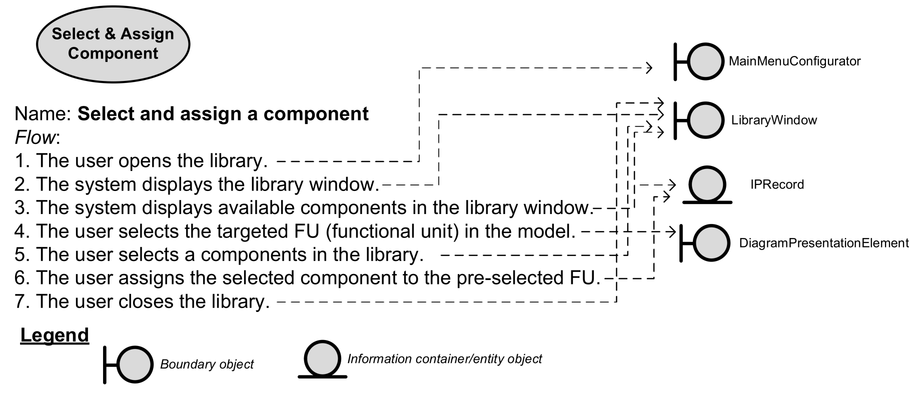
Actor	System designer
Precondition	The library window is already opened and displays available components in the library.
Postcondition	The library GUI closes.
Main path (M)	1. The designer clicks the "Close" button.

Use case 3: ***Select and assign component***

Actor	System designer
Precondition	The library window is already opened and displays available components in the library.
Postcondition	The chosen library component is assigned to selected model element.
Main path (M)	1. The designer selects a target model element. 2. The designer chooses certain component from a list of available library components. 3. The designer presses “Assign” button on the library main window.

Use case 4: ***Add component in the library***

Actor	System designer
Precondition	The library window is already opened and displays available components in the library.
Postcondition	The new component successfully adds to the library.
Main path (M)	1. The designer clicks “Add” button on the main library GUI window. 2. A new GUI appears with required input fields to get the information about the new component. 3. The designer provides information about the new component in the required input fields. 4. The design clicks “Add” button. 5. The new components adds into the library.



Select & Assign Component
MainMenuConfigurator
LibraryWindow
IPRecord
DiagramPresentationElement
Name: **Select and assign a component**
Flow:
1. The user opens the library.
2. The system displays the library window.
3. The system displays available components in the library window.
4. The user selects the targeted FU (functional unit) in the model.
5. The user selects a components in the library.
6. The user assigns the selected component to the pre-selected FU.
7. The user closes the library.
Legend
Boundary object
Information container/entity object

Figure 4.8: An analysis model of a "Select and assign component" use-case.

Use case 5: *Remove existing component from the library*

Actor	System designer
Precondition	The library window is already opened and displays available components in the library.
Postcondition	The selected component is removed from the library.
Main path (M)	1. The designer selects a certain component to be removed. 2. The designer clicks "Remove" button on the main library GUI window. 3. The component removes from the library database.

Next, an *analysis model* is built based on the developed use-case model, which describes the structure of the system. It consists of *analysis objects* that describe the logical implementation of the functional requirements that were previously identified in the use-case model. Here, the analysis model contains the possible stereotypical classes (analysis objects) responsible for certain roles in the system. These roles are: the boundaries (interface with the outside world i.e. screens/forms), the entities (information container) and the control objects (coordinators of the use case execution) [13]. Figure 4.8 illustrates a use case flow by considering the analysis objects. Other uses cases are developed in a similar fashion.

4.4.2 Implementation Approach

A *class diagram* describes the static structure of the system. The description contains the types of objects a system is composed of and the static relationships that exist among them. Here, a class diagram of the library is developed on the basis of the developed use case analysis model, as discussed previously. Later, the library is implemented according to the static structure described in the class diagram. Figure 4.9 shows the class diagram of the *SegBus* components library.

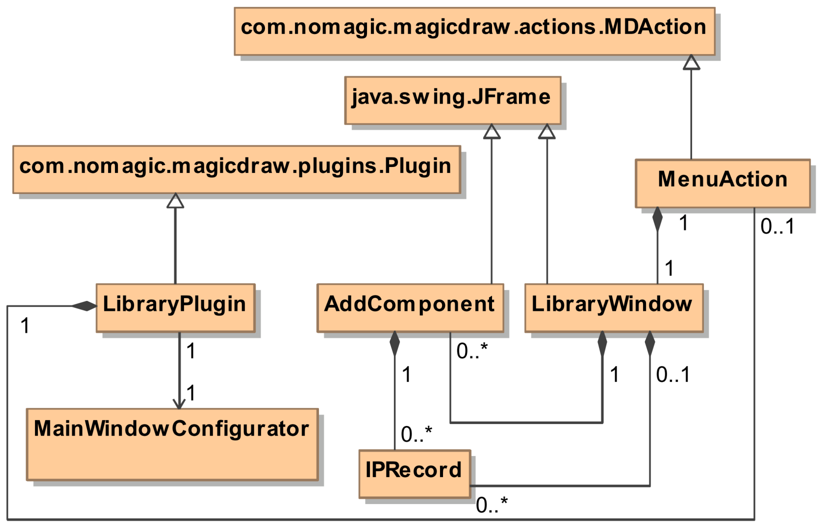


Figure 4.9: The class diagram of the *SegBus* components library.

Each class in Figure 4.9 is responsible for a particular role within library's functionality. Below, individual classes and their respective roles are discussed briefly.

JFrame class is part of the "Swing" package of the Java language [57] and it is used to produce a top-level window with a title and a border. Figure 4.10 illustrates the class structure of the *IPRecord* class. This class is a *java bean* - a class which allows access to its properties using dedicated *setter* and *getter* methods. An object of this class is used to hold important properties of a library component: name, feature size (technology), power consumption, storage location, and so forth. The description of each of its methods is described below.

IPRecord() - This is the constructor method of this class. It initializes the member variables with the passed values to it during its object initialization time.

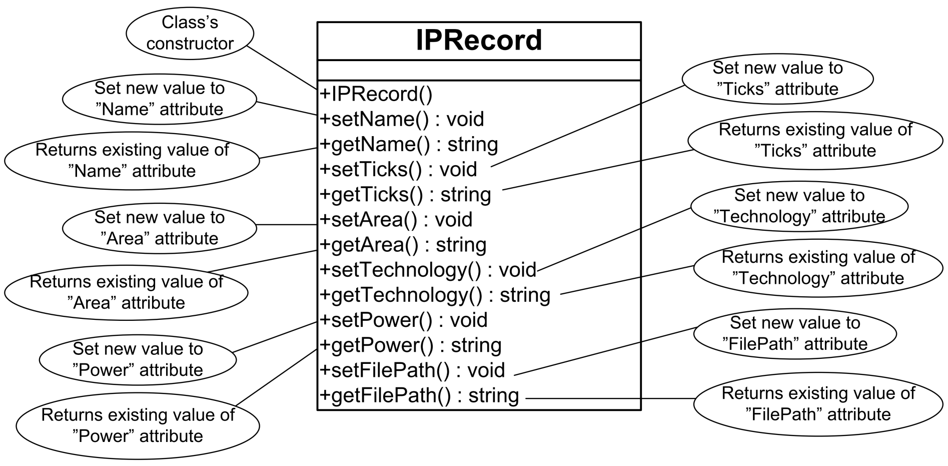

IPRecord
+IPRecord()
+setName() : void
+getName() : string
+setTicks() : void
+getTicks() : string
+setArea() : void
+getArea() : string
+setTechnology() : void
+getTechnology() : string
+setPower() : void
+getPower() : string
+setFilePath() : void
+getFilePath() : string
Class's constructor
Set new value to "Name" attribute
Returns existing value of "Name" attribute
Set new value to "Area" attribute
Returns existing value of "Area" attribute
Set new value to "Power" attribute
Returns existing value of "Power" attribute
Set new value to "Ticks" attribute
Returns existing value of "Ticks" attribute
Set new value to "Technology" attribute
Returns existing value of "Technology" attribute
Set new value to "FilePath" attribute
Returns existing value of "FilePath" attribute

Figure 4.10: *IPRecord* class structure and operations' description.

setName() - This method sets the "Name" of the library component. A string is passed to it as argument.

getName() - This method returns the name of the library component.

setTicks() - This method sets the "Ticks" of the component. Ticks represents total number of clock ticks this component generally takes to process one package.

getTicks() - This method returns the "Ticks" value of the component.

setArea() - This method sets the "Area" attribute of the component. Here, the area refers to total area to be occupied by this component on a particular fabrication technology.

getArea() - This method returns the area cost pertaining to this specific component.

setTechnology() - The *feature size* of the employed silicon technology for this library component is set by this method.

getTechnology() - This method returns the feature size of the component.

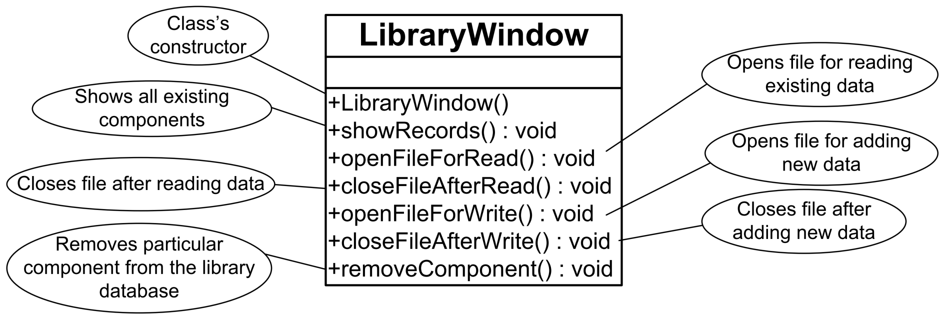
setPower() - The power consumption of this component is set by using this method. (Power consumption has not been considered in this research work. Here, the power field is introduced only to specify the power consumption of the respected IP component if it is known in advance. Further, it will be used in possible future work in the direction of system-level power estimation.)

getPower() - This method returns the power consumption of this component.

setFilePath() - This method sets the physical location of the component.

getFilePath() - This method returns the physical location of the component.

The structure and operations of the *LibraryWindow* class are depicted in Figure 4.11. This class is directly extended from *JFrame* class of the Java standard API [57] which is used to create a container for graphical components. The class is a central *window* which holds important graphical components of the library. The window is the starting point where the available library components can be seen in a graphical form. Additionally, the class provides the capability to add, remove and assign the library components to a model's element(s). A brief description of each of its member methods is given below.



LibraryWindow
+LibraryWindow()
+showRecords() : void
+openFileForRead() : void
+closeFileAfterRead() : void
+openFileForWrite() : void
+closeFileAfterWrite() : void
+removeComponent() : void
Class's constructor
Shows all existing components
Closes file after reading data
Removes particular component from the library database
Opens file for reading existing data
Opens file for adding new data
Closes file after adding new data

Figure 4.11: *LibraryWindow* class structure and operations' description.

LibraryWindow() - The constructor method of this class. The method initializes certain instance variables and displays the existing library components in graphical user interface (GUI). The constructor method calls the *showRecords()* method (described below) to get the existing library components and later displays them in a list form.

showRecords() - This method first gets the components from the library and then displays them on the GUI.

openFileForRead() - In Java, an *I/O stream* represents input source or output destination of a sequence of data [57]. The stream source and destination can be the file system, other programs and memory arrays. This particular method creates an instance of class *ObjectInputStream* [57] so that the library database can be read from the database file. The reason for using object-based streams for reading and writing data is due to rich support from available classes in Java. These classes (ObjectInputStream and ObjectOutputStream) enable the reading and writing of the Java primitive types and user-defined objects from/to the streams without any need further processing by the developer.

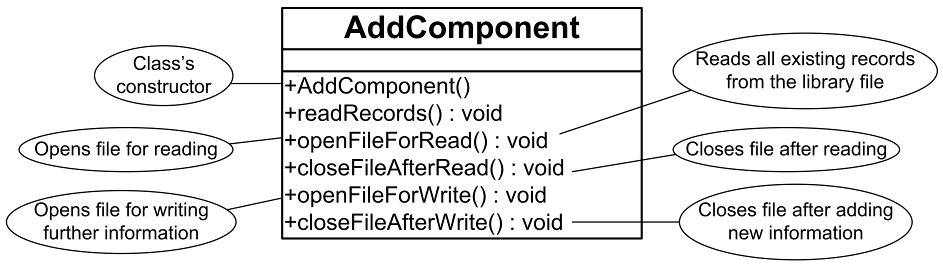
closeFileAfterRead() - This method closes the ObjectInputStream after reading the library components' records.

removeComponent() - This method removes the selected component from the library database.

openFileForWrite() - While removing selected component from the library database, this method is called by removeComponent() method. This method creates an instance of class *ObjectOutputStream* [57] to get access of the library database and the desired component is then removed from the database.

closeFileAfterWrite() - After removing the selected component from the library, the method closes the previously created ObjectOutputStream.

Figure 4.12 shows the *AddComponent* class structure. This class is also directly extended from *JFrame* class of the Java standard API like the previous one. The class provides a graphical window which holds other graphical components (text fields, etc.) that are used to add new components in the *SegBus* components library. Following a brief description of its member methods is presented.



AddComponent
+AddComponent()
+readRecords() : void
+openFileForRead() : void
+closeFileAfterRead() : void
+openFileForWrite() : void
+closeFileAfterWrite() : void
Class's constructor
Opens file for reading
Opens file for writing further information
Reads all existing records from the library file
Closes file after reading
Closes file after adding new information

Figure 4.12: *AddComponent* class structure and operations' description.

AddComponent() - This is the constructor method of this class which upon execution creates a GUI based on the JFrame class. The GUI allows the addition of new components in the library database. This is done by having important text fields in the GUI populated with information about the new component.

readRecords() - This method gets the existing records from the components' library and collect them into a *Vector* object [57]. This method is executed before adding new component into the library database.

addComponent() - This method inserts a new component into the library database. This is done by adding the new component in the previously created vector object, and then write back all the vector items into the database. The reason for initial reading the whole database file before adding the new component is to verify any duplicate items (identical to the new component) and thus avoid redundant entities in the library.

openFileForRead() - This method creates an instance of class *ObjectInputStream* so that the library database can be read to get the existing components.

closeFileAfterRead() - This method closes the *ObjectInputStream* after reading the library components from the database file.

openFileForWrite() - Before writing back all the records into the database file, an object of *ObjectOutputStream* is created with this method.

closeFileAfterWrite() - After successful writing back to the database file, this method closes the previously created *ObjectOutputStream* stream of data.

The *Open API* of the MagicDraw tool [56] is a collection of classes which allows writing user-defined plug-ins, create actions in the menus and toolbars, change UML model elements, etc. The *Plugin* class is part of the Open API and is the base abstract class for every plug-in of the tool. The plug-in under development must be extended from this class. Every plug-in has its own descriptor (discussed in section 4.4.3).

LibraryPlugin class is directly extended from "com.nomagic.magicdraw-.plugins.Plugin" class of the Open API. Figure 4.13 shows the class structure along with its member methods. Below the description of each method is discussed in brief.

init() - On *MagicDraw UML* startup, this overridden method is called which initializes the *SegBus* components library plug-in. It then registers the action method from *MenuAction* class to execute when the library plug-in is invoked to open the *components library GUI* during modeling.

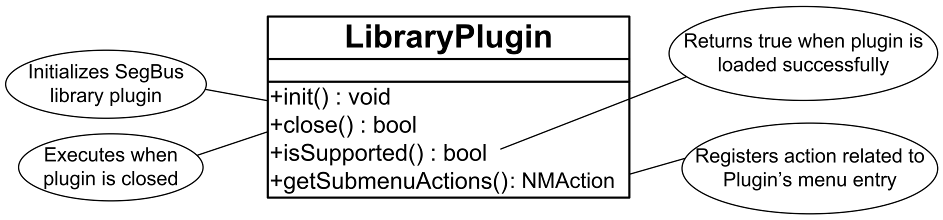

LibraryPlugin
+init() : void
+close() : bool
+isSupported() : bool
+getSubmenuActions(): NMAction
Initializes SegBus library plugin
Executes when plugin is closed
Returns true when plugin is loaded successfully
Registers action related to Plugin's menu entry

Figure 4.13: *LibraryPlugin* class structure and operations' description.

close() - This method is executed when the modeling tool shuts down. In the proposed library's case, it returns *true* to notify tool that the *SegBus* components library plug-in is ready to shut down too.

isSupported() - This method is called before initialization of every plug-in. If this method returns *false*, the plug-in will not be initialized. In the proposed library's case, this method returns *true* to allow the tool to initialize the *SegBus* components library plug-in.

getSubmenuActions() - This method is called by the *init()* method of this class. Here, this method first instantiates an instance of *Menu-Action* class (discussed below) and an instance (called as *category*) of *ActionsCategory* class from the Open API. Second, it adds the *Menu-Action*'s instance into the category. Finally, it instantiates an instance of *MainMenuConfigurator* (discussed below) and forwards it to the newly created category for further processing.

Figure 4.14 shows the static structure and member functions of the *Main-MenuConfigurator* class. The *action manager* of the tool is responsible for managing the actions and categories. It has a list of categories where it registers different actions by shortcuts and IDs.

The *MainWindowConfigurator* class implements *AMConfigurator* interface from the *com.nomagic.actions* package of the *Open API*, which in turn, is used to configure the action manager. The *AMConfigurator* is a general-purpose configurator in the MagicDraw UML tool which is widely used for configuring menus, toolbars, browser and shortcuts for different diagrams used in the tool [56]. A separate menu for the library in the tool's main menu bar is created with the help of this class. Below, a brief description about the functionality of each of its member methods is discussed.

MainMenuConfigurator() - This is the constructor method of the class which is used to introduce a new menu in the tool and registers an appropriate action (the *MenuAction* in this case) whenever the menu is clicked.

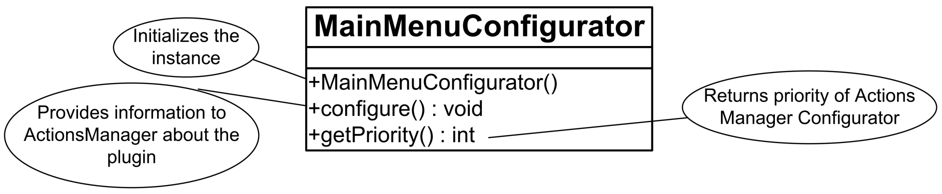


Figure 4.14: *MainMenuConfigurator* class structure and description of its methods.

configure() - This is an overridden method of this class for the *AMConfigurator* implementation. This method provides information to the action manager about the *SegBus* components library plug-in.

getPriority() - All configurators in the MagicDraw UML are sorted by priority before configuration [56]. The priority is very important if one configurator expects actions from other configurators. Here, the *SegBus* components library plug-in uses the medium priority for the MainMenuConfigurator class and this method (getPriority()) returns the priority of the created menu to the MagicDraw UML.

The *MDAction* class belongs to the "com.nomagic.magicdraw.actions" package of the Open API. It provides methods which can be overridden to offer desired functionality whenever a related action is occurred within the tool. The *MenuAction* class is directly extended from the "MDAction" class. Figure 4.15 depicts the structure of this class and the description of each of its member methods is mentioned below.

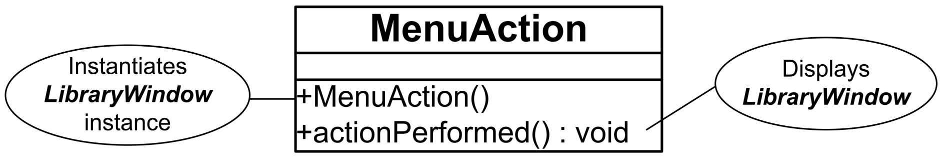

MenuAction
+MenuAction()
+actionPerformed() : void
Instantiates LibraryWindow instance
Displays LibraryWindow

Figure 4.15: *MenuAction* class structure and operations' description.

MenuAction() - This is the constructor method of this class. This method sets important attributes e.g. name, ID, etc. In addition, it creates an instance of the *LibraryWindow* class so that the *SegBus* components library's main GUI can be displayed in the modeling tool, when the user clicks on the menu item.

actionPerformed() - In the modeling environment, when the designer intends to open the *SegBus* components library, the designer goes to main toolbar of the modeling tool and clicks on a separate menu for the components library named as “Library”. The library menu contains the item “*SegBus* Components Library”. When the designer clicks on this menu item, an *ActionEvent* is generated and passed on to *actionPerformed()* method. The method then makes the library's GUI visible, and hence, available to the designer.

4.4.3 Plug-in Setup

Plug-ins are the only way to add or change functionality in the MagicDraw tool [56]. This thesis develops the *SegBus* components library in a plug-in form such that it can be effectively utilized as a pool of reusable IP components by the tool.

A plug-in must contain: a directory, compiled Java files packages into *jar* files, a plug-in descriptor file and optional files to be used by that plug-in. Apache Ant [55] is used to compile the source code of the library together with the provided class library of the tool, and further package them into a *jar* file (described below). Following that, the script is shown below, which has been used to compile and package the source code into a *jar* file.

```
<project name="SBLibrary" basedir="." default="main">
  <property name="lib.dir"     value="../../lib"/>
  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar"/>
  </path>
  <target name="clean">
      <delete dir="build"/>
  </target>
  <target name="compile">
      <mkdir dir="build/classes"/>
      <javac srcdir="src" destdir="build/classes"
             classpathref="classpath"/>
  </target>
  <target name="jar">
      <mkdir dir="build/jar"/>
      <jar destfile="build/jar/SBLibrary.jar"
           basedir="build/classes">
          <manifest>
              <attribute name="Main-Class"
                         value="SBLibrary.LibraryPlugin"/>
          </manifest>
      </jar>
  </target>
</project>
```

The MagicDraw UML, on every startup, scans the plug-ins directory and looks for further sub-directories. If a sub-directory contains a plug-in descriptor file (named as “*plugin.xml*”), then the *plug-in manager* of the MagicDraw UML reads it. Next, if requirements specified in the descriptor file are satisfied then the plug-in manager executes the specified class by calling its *init()* method. Therefore, the specified class (*LibraryPlugin* in this case) must be derived from "com.nomagic.magicdraw.plugins.Plugin" class in order to successfully get triggered by the plug-in manager.

The presented research work thus performed all the necessary steps to make up the appearance of the library as an executable plug-in. To achieve this, a directory is created (*$(MagicDraw)\plugins\SBLibrary*). Next, compilation is performed that builds a packaged *jar* file (*$(SBLibrary)\build\jar\SBLibrary.jar*) based on its general principles. Then, a descriptor file has been written. The contents of the descriptor file are shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
 <plugin id="SegBus.Library.Plugin"
        name="SegBus IP Library Plugin"
        version="1.0" provider-name="Moazzam"
        class="SBLibrary.LibraryPlugin">

 <requires>
   <api version="1.0"/>
 </requires>

 <runtime>
   <library name="build/jar/SBLibrary.jar"/>
 </runtime>
</plugin>
```

Interested readers can find out the detailed semantics about various elements and related attributes used in the above descriptor file in "*Open API* user guide" [56].

In general, the employment of reusability at different stages of the platform-based design processes not only reduces the design time, but comprehensively supports the automation between different sub-tasks in the design processes. As discussed previously, the continuous evolution of computer-aided design (CAD) tools and reusability in design methodologies are the key enablers for the successful transition to the platform-based design. Similarly, in the context of the *SegBus* platform, this research considers the development of this library as a significant contribution in continuing efforts towards building a unified framework for designing and implementing applications on to the *SegBus* platform. This is the key contribution of the work presented in this thesis.

4.5 Summary

This chapter introduced a DSL for modeling and mapping of an application onto an instance of the *SegBus* communication platform. The chapter described in the form of graphical elements the principal structural elements of the platform with their structural relations and the related DSL customization.

The DSL provides an environment where a designer can model platform and associate it with the application components using different configurations in an automated manner. The DSL also helps in model transformation at later stages of development process. Moreover, the DSL supports designers in modeling phase whereby none of the structural constraints are violated. The validation suite embedded in DSL helps designers to rectify the problems in the model of the system and correct them accordingly.

Together with DSL, a library of components for the *SegBus* platform has also been presented , named as, *SegBus Components Library*, which is a collection of reusable hardware/software components stored in a centralized library. This technique enables addressing evolving challenges with the current trends in embedded systems development by applying IP reusability during the design process.

Chapter 5

SegBus Emulator

This chapter presents an emulator program, the *SegBus emulator*, and a performance estimation technique for the *SegBus* platform. The technique enables the assessment of the performance of any specific application mapped to a particular platform configuration, modeled using the *SegBus* DSL. The chapter presents methods to transform the Package Synchronous Data Flow

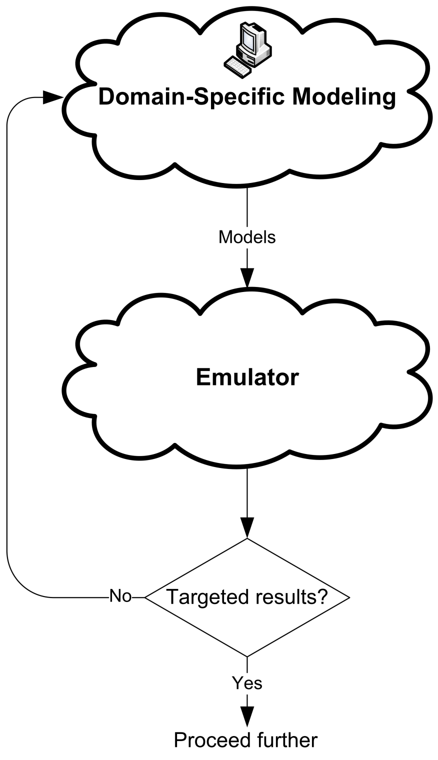


Figure 5.1: High-level view of design entry to *SegBus* emulator.

(P-SDF) and Platform Specific Model (PSM) of the application into Extensible Markup Language (XML) schemes using the modeling tool. The chapter also shows that how the generated XML schemes can be utilized by the emulator program to get the execution results. The technique enables the performance estimation of the mapped application on a number of different platform configurations during the early stages of the design process. Figure 5.1 shows a high-level view of design transition between *SegBus* DSL and *SegBus* emulator.

5.1 Basic Concepts

In the proposed *SegBus* emulator approach, the following considerations apply to build the emulator as a close match to the *SegBus* communication platform and to the application execution.

1. The schedule of the application is extracted from the P-SDF and implemented within the arbiters, providing the correct sequencing among processing and transfers.

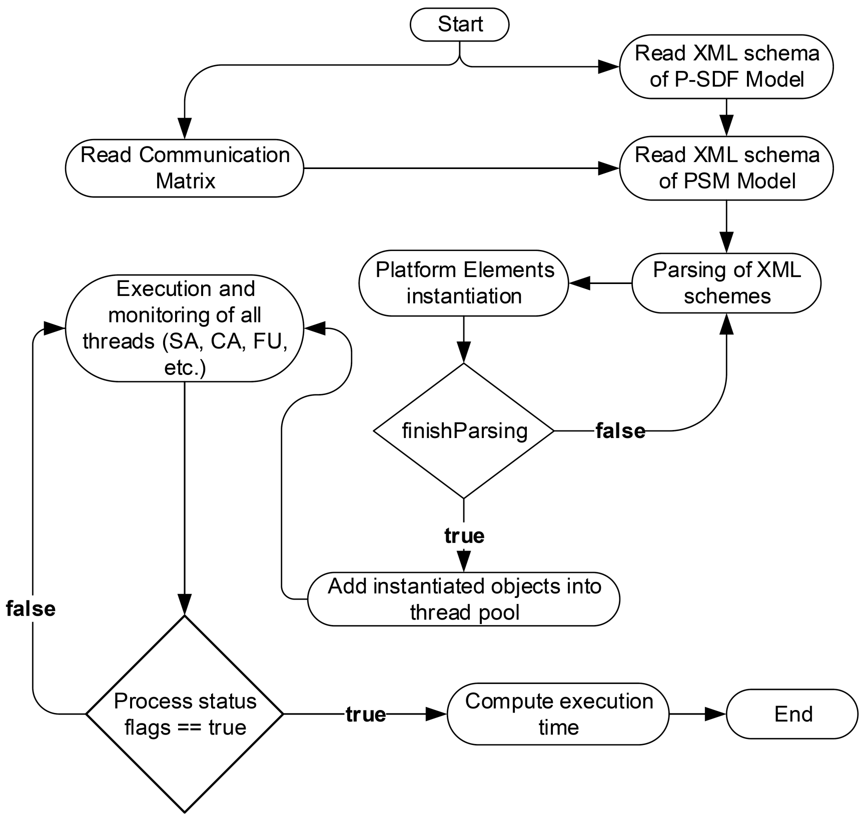
2. As for the moment the designer is not interested in the actual operational results, therefore the ***FU***s are modeled as counters, performing for an established duration. The count values of the counters will stand as a "processing" time associated with each ***FU***. This time (in the form of clock ticks - CT) will be extracted from the P-SDF model where each ***FU*** is assigned a CT value. A CT value represents that how many clock ticks are required by a ***FU*** to process one package.

3. The performance measurements (execution time) are established with respect to the starting moment of the emulation process. While for individual processes this might introduce errors in measurement (as certain modules have to wait until data is present in order to start operating), this does not affect the overall application time performance - which is the main target in this study. The experimental results (discussed in detail in Chapter 7.2) show that the proposed emulator results (representing overall application time performance) only deviate within 5-7% of the simulation results over real platform instances and thus validates this argument.

4. The emulator is equipped with an array of flags - "Process Status Flags", each element here corresponding to one ***FU*** process of the application. When a process finishes its activities and related transfers, the appropriate flag is raised.

5. During the execution of the application on the emulated platform, monitoring activities are executed to measure the execution times (clock ticks) of the ***FU***s, ***SA***s and of the ***CA***.

6. The system emulation is considered finished when all the flags described above are high, and there is no activity to execute within any of the platform's ***SA***s or ***CA***.



Start
Read XML schema of P-SDF Model
Read Communication Matrix
Read XML schema of PSM Model
Platform Elements instantiation
Parsing of XML schemes
Execution and monitoring of all threads (SA, CA, FU, etc.)
finishParsing
false
true
Add instantiated objects into thread pool
false
Process status flags == true
true
Compute execution time
End

Figure 5.2: Operating flow of the emulator.

5.2 Model Transformations

The first phase for performing the emulation of any *SegBus* configuration modeled in DSL is to transform the models into XML schemes so that the configuration can be used by the emulator program for further analysis. The choice of XML-based approach allows models to be readable by both humans and machines (*SegBus* emulator in this case). This approach provides ease in information exchange between concerned entities in the framework and allows for a smooth flow.

The emulator application is written in Java language [57] due to its rich collection of classes for handling XML schemes, and classes for implementing multi-threaded application (discussed in section 5.4). The code generation engine provided by the modeling tool does provides the capability to transform model(s) into XML schema as per the M2T specification [59].

Before transforming models to XML schemes, a *code engineering set* needs to be introduced in the modeling tool for each model where the designer specifies the required type of transformation i.e. Model-to-Model, Model-to-Text (as in this case), etc. The code engineering set consists of a set of model elements from a respective source model whose XML contents need to be generated during transformation process.

Targeting the proposed approach of this thesis, two distinct code engineering sets are made for the already modeled target system (one for PSM and the other one for P-SDF). The code engineering set representing PSM contains the platform elements (SAs, CA, BUs, etc.). While the other code engineering set representing P-SDF contains all the application components in the form of processes (P0, P1, etc.). A directory is also specified where the generated XML schemes are saved. After applying transformation on desired P-SDF and PSM models, the required XML schemes are obtained in the mentioned directory.

The generated XML consists of a *schema* element and a number of sub-elements, in the form of *complexType* and *element* types. Each complex type represents a platform element (***CA***, ***SA***, etc.) or application component (P0, P1, etc.). The *name* attribute of each complex type shows the name of the element. In addition, each complex type may contain sub-elements. Figure 5.3 shows the hierarchical structure of the platform elements. At the top level is the *SegBusPlatform* itself consisting of *Segment*(s) and exactly one ***CA***. Every segment is composed of at least one ***FU***, and exactly one ***SA***. Each segment is connected with other neighboring segment through a ***BU***. Each ***FU*** must contain at least one *Master* or one *Slave*. A ***FU*** may contain a combination of both master(s) and slave(s). The generated XML also follows the same hierarchy in representing platform entities in XML form.

Next, an XML snippet is shown from the example P-SDF model of Figure 5.4 after transformation, consisting of processes *P0*, *P1*, *P3* and *P4*, and their relative transfers to other processes.

```
<xs:complexType name="P0">
   <xs:sequence>
      <xs:element name="P1_576_1_250" type="P1"/>
      <xs:element name="P8_576_1_250" type="P8"/>
   </xs:sequence>
</xs:complexType>
```

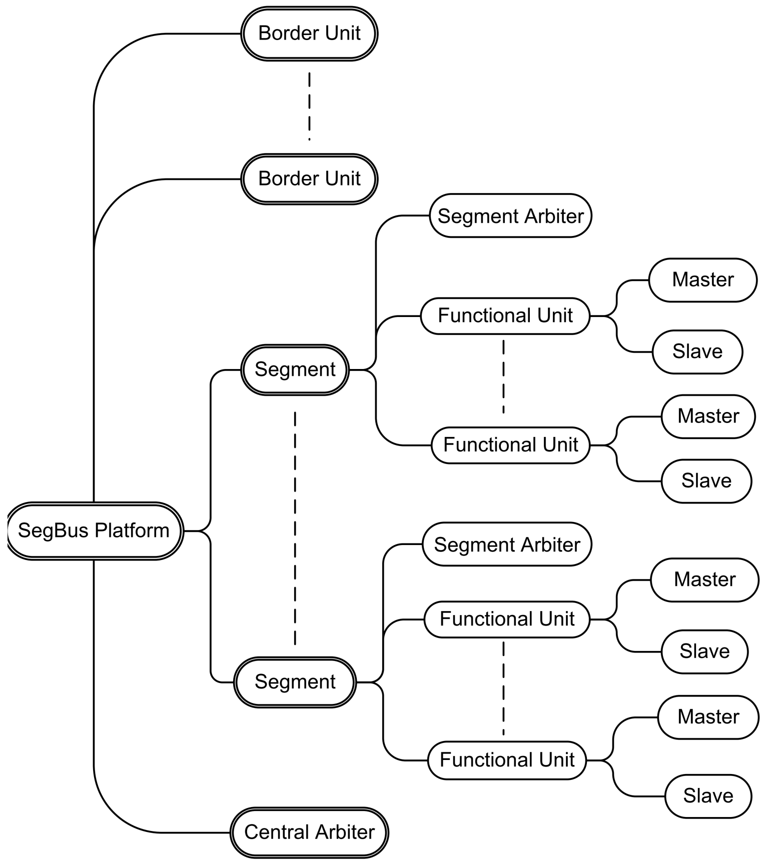


Figure 5.3: Hierarchical structure of the SegBus profile elements.

```
<xs:complexType name="P1">
   <xs:sequence>
       <xs:element name="P2_540_2_250" type="P2"/>
       <xs:element name="P3_36_3_250" type="P3"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="P3">
   <xs:sequence>
       <xs:element name="P4_36_4_500" type="P4"/>
       <xs:element name="P10_36_4_500" type="P10"/>
       <xs:element name="P11_540_4_500" type="P11"/>
       <xs:element name="P5_540_4_500" type="P5"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="P4">
   <xs:sequence>
       <xs:element name="P5_36_5_250" type="P5"/>
   </xs:sequence>
</xs:complexType>
```

Below is an extract from the XML snippet of an example PSM model after transformation, representing the *SegBus* platform instance named as *SBP* with three segments in a linear topology configuration, ***CA*** and ***BU***s as child-elements: "*Segment 0*" element with its child-elements (process *P0, P1, P2, P3, P8, P9* and *P10*; segment arbiter *SA0*; and ***BU*** *buRight*); "*Segment 1*" element with its child-elements (process *P5, P6, P7, P11, P12, P13* and *P14*; segment arbiter *SA1*; and ***BU*** *buRight* and *buLeft*).

```
<xs:complexType name="SBP">
   <xs:all>
      <xs:element name="segment0" type="Segment0"/>
      <xs:element name="segment1" type="Segment1"/>
      <xs:element name="segment2" type="Segment2"/>
      <xs:element name="ca" type="CA"/>
      <xs:element name="bu12" type="BU01"/>
      <xs:element name="bu23" type="BU12"/>
   </xs:all>
</xs:complexType>

<xs:complexType name="Segment0">
   <xs:all>
      <xs:element name="p0" type="P0"/>
      <xs:element name="p1" type="P1"/>
      <xs:element name="p2" type="P2"/>
      <xs:element name="p8" type="P8"/>
      <xs:element name="p9" type="P9"/>
      <xs:element name="p10" type="P10"/>
      <xs:element name="p3" type="P3"/>
      <xs:element name="arbiter" type="SA0"/>
      <xs:element name="buRight" type="BU01"/>
   </xs:all>
</xs:complexType>

<xs:complexType name="Segment1">
   <xs:all>
      <xs:element name="buRight" type="BU12"/>
      <xs:element name="buLeft" type="BU01"/>
      <xs:element name="p5" type="P5"/>
      <xs:element name="p6" type="P6"/>
      <xs:element name="p7" type="P7"/>
      <xs:element name="p11" type="P11"/>
      <xs:element name="p12" type="P12"/>
      <xs:element name="p13" type="P13"/>
      <xs:element name="p14" type="P14"/>
      <xs:element name="arbiter" type="SA1"/>
   </xs:all>
</xs:complexType>
```

5.3 Setup for Emulation

Parsing is a process in which text data is *analyzed* to extract meanings and information from a machine. XML parsing is performed on XML documents and schemes to get the contained information from the provided readable XML.

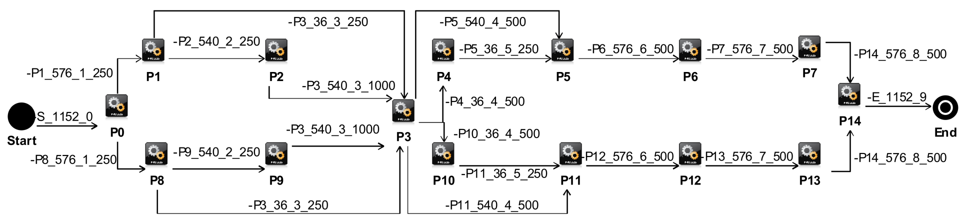


Figure 5.4: P-SDF model of an example application.

The next phase of the design methodology is to parse the generated XML files and build the structure of platform and allocation of resources on it within the emulator. The *DocumentBuilderFactory* and *DocumentBuilder* classes from the *javax.xml.parsers* package were utilized in order to create *XML document* for further parsing. The *parse* method of the *DocumentBuilder* class returns XML *Document* object, when the designer supplies the generated XML file(s).

The emulator program handles the communication matrix in two different possible ways. In the first method, it reads a text file which contains the communication matrix entries in text form. The first method is used when the P-SDF model do not provide information about the amount of data to be transferred in each transaction. In the second method, the emulator program create a communication matrix from the information that are extracted from the P-SDF model. The second method is used when the transactions in the P-SDF model are modeled in a way representing the amount of data to be transferred in each transaction.

The reason for these two different ways for handling the communication matrix is that, initially when this research work was started, the only way to provide communication matrix to the emulator was in the form of a text file. Later, as time passed by, the emulator functionality were enhanced to further extent so that it can now fetch and create the communication matrix from the P-SDF model and the legacy functionality is kept intact for possible future use. Generally, the second method of building communication matrix is used when the information about the data transfers are already contained in the P-SDF model. Figure 5.2 depicts this feature in a detailed manner in graphical form.

As discussed above, the emulator program builds the communication matrix by extracting transactions between the processes in P-SDF model. Based on the matrix, the (independent) *SBTool* utility [19] finds the optimal device allocation solution, given the platform specifics (the number of segments). The optimal allocation results from the SBTool utility are treated as *allocation proposal* which helps the allocation of application processes on different segments of the platform's instance considering the designer obtains the desired performance scores after emulation. Otherwise, the designer modifies the processes' allocations accordingly, if necessary.

The emulation process is based on both P-SDF and PSM models. The P-SDF model provides information about interaction between application processes with required data items and other useful parameters, while the PSM model represents the placement of each application process on different segments of the platform. Hence, the emulator program parses the XML of both models to be later used for emulation. During the parsing process, the emulator extracts following information from the P-SDF model:

- Number of application processes.
- Data transfers from each process.
- Ordering of transfers.
- Clock ticks to be consumed by each process while processing one package.

The emulator stores the above information in temporary variables and arrays. Figure 5.4 shows the P-SDF model of the example application (briefly discussed in Chapter 7). For instance, the name attribute from one of the *element* from process *P0*, that is, "P1_576_1_250" represents a transfer from process *P0*. The "_" character serves as the separator between the entities. The first entity "*P1*" represents the target process of this transfer; the second entity "576" is the number of data items to be transferred; the third entity "1" is the sequencing order and the last entity "250" is the number of clock ticks a process needs to consumed before sending each package.

Furthermore, the emulator extracts following information from the PSM model and stores in a number of variables and arrays inside the emulator, too:

- Platform's communication elements (platform instance (**SBP**), ***CA***, ***BU***s, etc.)
- Number of segments in the platform.
- Number of border units based on platform geometry.
- Placement of application processes on different segments.

When the parsing process is finished for the XML of the P-SDF model, the emulator iterates in the previously populated arrays, instantiates objects of the required ***FU***s and pass them necessary information. This necessary information contains number of data items to be transferred, destination processes, relative ordering, clock ticks a process needs to be consumed before sending a package and placement in the specific segment. The *constructor* method of the ***FU*** class analyzes the passed information to it and instantiates the required number of objects of *masters* and *slaves*, which later to be executed as threads during emulation.

The emulator has been programmed in a way to exhibit the behavior of an actual platform instance. The functionality and behavior of each platform element (***SA***, ***CA***, ***BU***, etc.) are programmed and stored in individual Java source files. A number of monitoring statements are introduced in different sections of ***SA***, ***CA*** and ***BU*** codes. These monitoring statements count clock ticks involved in any transfer, either intra-segment or inter-segment. The *arbitrate* method in ***CA*** and ***SA*** source code performs arbitration and called by the emulator application several required number of times during execution. The method also counts how many clock ticks have been consumed for any particular transfer at different stages of the operation.

At the ***SA*** level, certain statements are setup in an *arbitrate* method (discussed in next section) to count requests coming from the application processes. Separate counters are also setup inside source code to count both kinds of requests (intra and inter-segment). These statements help us later to analyze the configured system and provides means to take optimal decision according to needs. In case of inter-segment transfers, there exist another set of counters to count how many packages transferred to left and right side ***BU***s.

At the ***CA*** level, monitoring statements in *arbitrate* method count the number of clock ticks ***CA*** consumed while *setting* and *resetting* related grant signal in response to inter-segment requests. The monitoring statements at ***BU*** level counts how many packages received from, and transferred to, left and right-side segment. It also counts total number of clock ticks during all transfers.

During the parsing process of XML for the PSM model, the emulator application first looks for the *SegBus* platform instance in the XML document, analyzes its structure by counting how many segments and ***BU*** it contains as child nodes. It instantiates an object of platform instance, ***CA***, required number of ***BU***s and saves the references (discussed below). Later on, it looks for the elements in XML document, which represent segments. It analyzes the structure of each segment, instantiates one ***SA*** and required number of ***FU***s associated with any particular segment and pass the object reference of segment to left/right ***BU***(s).

The emulator application maintains a number of lists each for different communication (***CA***, ***SA***, ***BU***, etc.) and application (***FU***) components. Whenever it encounters specific element in the XML document, it instantiates an object of the relevant class and adds it to corresponding list. For instance, if the emulator program finds an element representing a ***BU*** in the XML document, it instantiates an object of class *BU* by calling the constructor and passing the necessary values and adds the object to a list that holds only ***BU*** objects.

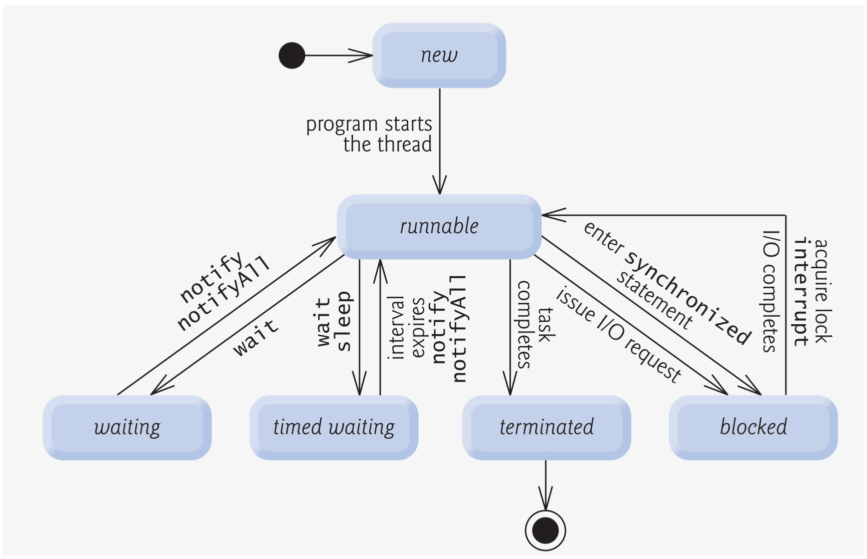


Figure 5.5: Thread life cycle in different states [9].

5.4 Implementation Approach

The microprocessor in a personal computer (PC) executes computer program instructions in a sequential order. On the other hand, hardware devices perform their dedicated functionality in parallel with other devices. The main challenge of emulator development in this research work was to transform the parallel behavior of hardware elements associated with platform entities into some special form that can be run on the microprocessor-based PC and exhibit the correct characteristics of the hardware devices. A variety of programming and execution models exist to model concurrency in computer programs such as POSIX [71], OpenMP [72], MPI [73], etc. Each one of them has some particular characteristics which make them suitable for a particular context and *multi-threading* is one of them.

Multi-threading is not a new idea and has been in existence for many years. Generally, every running program on a PC is called a *process*. Multi-threading is the task of creating a new *thread* of execution within an existing process rather than starting a new process to begin function and running them concurrently. All the threads in a process share the same allocated memory and execute independently with other threads. The “concurrent” execution of threads within the same process is often considered as a more efficient use of the resources of the PC. Multi-threading employs time-division multiplexing to execute threads concurrently. Threads are obtained from the pool of available ready-to-run threads and execute on the available microprocessor(s).

Thread Life Cycle. During the life cycle of a thread, it lives in one of different *thread states* [9]. Figure 5.5 shows the life cycle of a thread. A thread starts its life cycle in *new* state whenever it is created. After thread creation, the thread moves to *runnable* state and is executed by the processor. Later, a thread moves to *waiting* state while it waits for other threads to complete their tasks. The thread remains in this state until it is notified. In addition, a *timed waiting* is another form of waiting state. A thread remains in this state waiting for a certain time interval to finish. Furthermore, a thread moves to *blocked* state when the thread tries to complete a task and the task cannot be completed right away. It temporarily waits in the blocked state until it is ready to complete the task. Lastly, a thread moves to *terminated* state when it completes it required tasks.

The thesis employs Java's multi-threading feature in the proposed emulator because of Java's rich support for implementing multi-threaded applications. In the proposed emulator approach, all the classes related with emulator (emulator engine and source files related to platform) run as threads during execution. Each class implements the *Runnable* interface from *java.lang* package by introducing a specific *run()* method. The method executes when emulation starts and performs dedicated functionality.

Class descriptions. In Figure 5.6, a (simplified) class diagram of the emulator program is illustrated with the most important classes and their relationships. The diagram is simplified by omitting class attributes and methods, for the purpose of increasing clarity.

Figure 5.7 shows the static structure of class ***SBP***. This class is used to instantiate a logical instance of the *SegBus* platform in the emulator. Here, attributes of class are omitted from the shown structure to establish simplicity. This class consists of 3 major methods, as described below.

SBP() - This is the constructor method for the class which is responsible for preparing and instantiating the class's object and initializing various objects and constituent members.

setMonitor() - This method creates an array of status flags for all the ***FU***s in the platform's instance, and instantiates a *MonitorClass* object and forward to it the created array of status flags for further operations.

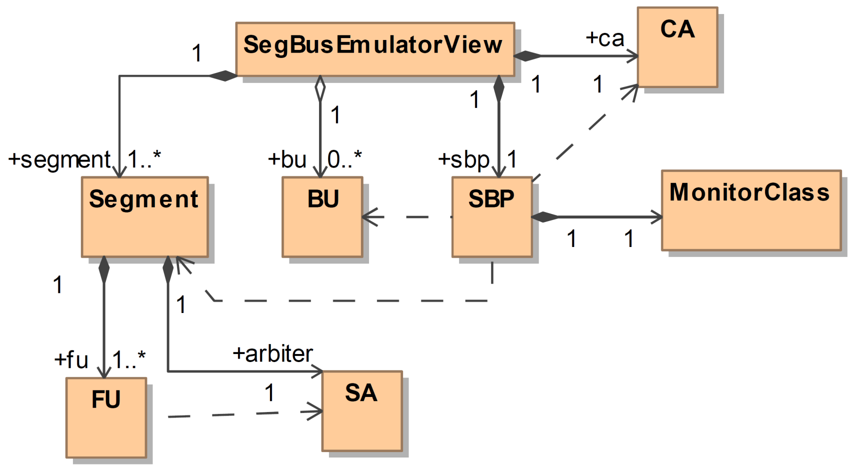


Figure 5.6: Class diagram of the emulator application.

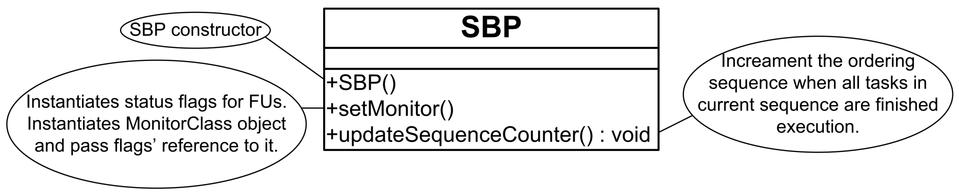

SBP
+SBP()
+setMonitor()
+updateSequenceCounter() : void
SBP constructor
Instantiates status flags for FUs. Instantiates MonitorClass object and pass flags' reference to it.
Increament the ordering sequence when all tasks in current sequence are finished execution.

Figure 5.7: ***SBP*** class operations' description.

updateSequenceCounter() - This method updates the under-execution ordering sequence counter, when all the task associated with current sequence number has been finished.

Figure 5.8 depicts the class ***SA*** static structure including with the specification of its methods. Following a brief description of each method is presented.

SA() - As the name suggest, this method is the class constructor, which prepares the instance of this class. In addition, it also initializes important objects and constituent members.

run() - This method starts execution as soon its thread moves to runnable state by the operating system. The core functionality of ***SA*** are part of this method.

SetReqs() - This method sets incoming request from a master for bus grant. The request is considered for bus grant in the next round of arbitration.

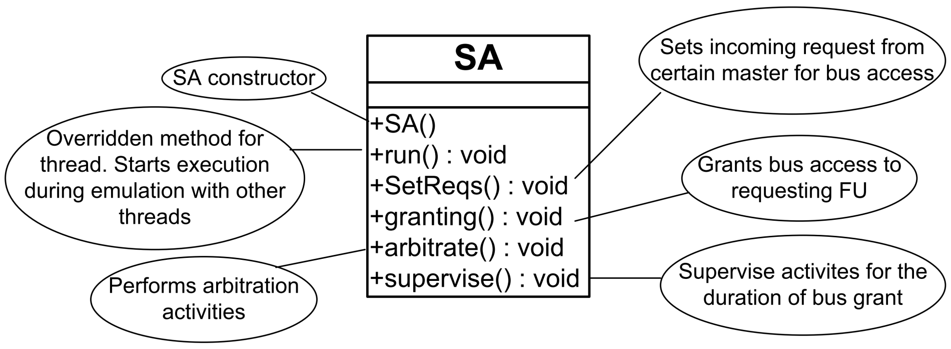

SA
+SA()
+run() : void
+SetReqs() : void
+granting() : void
+arbitrate() : void
+supervise() : void
SA constructor
Overridden method for thread. Starts execution during emulation with other threads
Performs arbitration activities
Sets incoming request from certain master for bus access
Grants bus access to requesting FU
Supervise activites for the duration of bus grant

Figure 5.8: ***SA*** class operations' description.

granting() - This method grants bus access to a certain requesting master for data transfer to intra-segment slave(s) or to inter-segment slave(s). In case of inter-segment transfer, the ***SA*** also notifies appropriate ***BU*** with respect to the direction of data transfer, and the master later fills the buffer of the relevant ***BU*** accordingly.

arbitrate() - This method *arbitrates* data transfer activities inside a segment. First, it checks whether is there any request for bus access from any master attached with concerned segment and take appropriate *granting* action. Second, if a master is already holding the bus grant, then it takes appropriate *supervisory* action.

supervise() - This method *supervise* data transfers, when a master is been issued a bus grant. It checks the status of the ongoing data transfers and its duration. If the master has already completed data transfer or the duration of the bus grant is already passed, then this method take further actions to set/reset certain signal so that the bus could be available for further requests.

The static structure of class ***CA*** is depicted in Figure 5.9. This class contains 4 major methods whose brief description is mentioned below.

CA() - The constructor method of the class which prepares newly created object of this class for the proper usage. It also initializes constituent members with appropriate values.

run() - As described previously, this method starts execution as soon its thread moves to runnable state by the operating system. The core functionality of ***CA*** are part of this method and it is repeatedly called *arbitrate* method pertaining to this class to perform its dedicated operation.

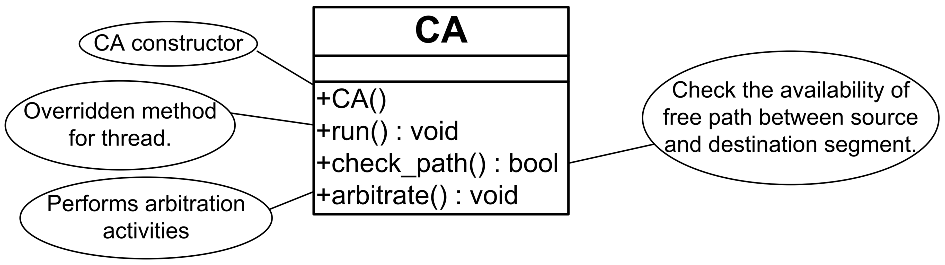

CA
+CA()
+run() : void
+check_path() : bool
+arbitrate() : void
CA constructor
Overridden method for thread.
Performs arbitration activities
Check the availability of free path between source and destination segment.

Figure 5.9: ***CA*** class operations' description.

check_path() - This method finds out the possibility of free path between any source and destination segment, when inter-segment communication is requested from the ***CA***The method is repeatedly called in each arbitration round by *arbitrate* method (discussed below) until the path is free for serving pending requests.

arbitrate() - This is the main method of ***CA*** responsible to perform arbitration related tasks and it is executed every time when ***CA*** thread moves to *runnable* state, and hence, it performs an arbitration round. When an inter-segment communication request is made from a certain segment, it checks the availability of free path between source and destination segments. If the path is available, it grants the path to requested segment, otherwise it move the request into a queue for pending requests and are considered for grant in the next arbitration round. In addition, it resets various grant signals pertaining to each involved segment as soon the inter-segment communication is gradually completed in a segment-by-segment manner.

Figure 5.10 shows the ***BU*** class structure. The class contains 6 major functions and a brief description about them is mentioned below.

BU() - This is the (overloaded) constructor method of the ***BU*** class. During execution, it sets the instance name and initializes numerous important objects and variable which are useful for its correct execution.

run() - The implementation of *run()* method from the *Runnable* interface. As described earlier, it is executed when its thread moves to *runnable* state.

setDepth() - This method is used to set the depth of FIFO queue inside a ***BU*** object. The FIFO depth is based on the package size to be used platform-wide during emulation.

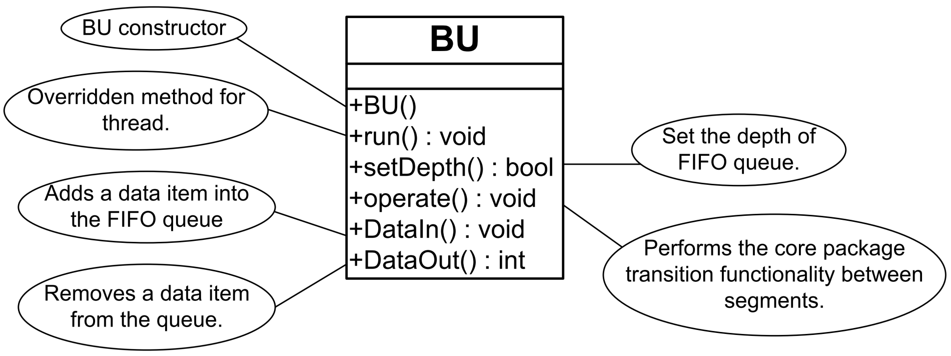

BU constructor
Overridden method for thread.
Adds a data item into the FIFO queue
Removes a data item from the queue.
BU
+BU()
+run() : void
+setDepth() : bool
+operate() : void
+DataIn() : void
+DataOut() : int
Set the depth of FIFO queue.
Performs the core package transition functionality between segments.

Figure 5.10: ***BU*** class operations' description.

operate() - This is a core function of the ***BU*** class and it performs all the major tasks of the ***BU***. First, it receives package from a particular master in one segment and copy the received package into the FIFO buffer. Second, it requests appropriate neighboring segment to get the package and then act accordingly. If the package in the ***BU*** belongs to one of the slave in the neighboring segment, the ***SA*** in the neighboring segment informs the targeted slave to get the package, otherwise the neighboring segment copies the package from one ***BU*** to another ***BU*** in a specific direction and this process continues until the package get to its destination segment.

DataIn() - This method inserts the received package data into the FIFO buffer.

DataOut() - This method removes the package data from the FIFO buffer.

The static structure of the *SegBusEmulatorView* class is illustrated in Figure 5.11. The class performs the important functions of the emulator program. First, it contains methods to read and parse the XML schemes of the P-SDF and PSM models, and to set-up the emulation process. Second, this class is responsible for creating and setting up a thread pool for the threads to be executed. Below a brief description about each method of this class is presented.

SegBusEmulatorView() - The constructor method of this class which creates user interface of the emulator application, initialize certain objects and important variables.

ReadPSDF() - This method is used to read and parse XML schema of the P-SDF model of the target application.

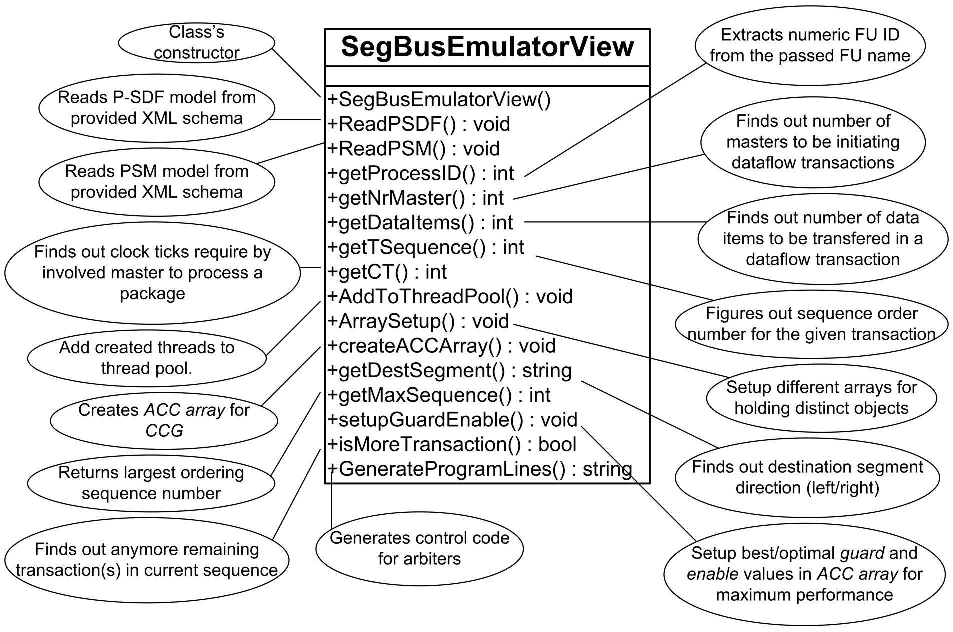

SegBusEmulatorView
+SegBusEmulatorView()
+ReadPSDF() : void
+ReadPSM() : void
+getProcessID() : int
+getNrMaster() : int
+getDataItems() : int
+getTSequence() : int
+getCT() : int
+AddToThreadPool() : void
+ArraySetup() : void
+createACCArray() : void
+getDestSegment() : string
+getMaxSequence() : int
+setupGuardEnable() : void
+isMoreTransaction() : bool
+GenerateProgramLines() : string
Class's constructor
Reads P-SDF model from provided XML schema
Reads PSM model from provided XML schema
Finds out clock ticks require by involved master to process a package
Add created threads to thread pool.
Creates ACC array for CCG
Returns largest ordering sequence number
Finds out anymore remaining transaction(s) in current sequence
Generates control code for arbiters
Extracts numeric FU ID from the passed FU name
Finds out number of masters to be initiating dataflow transactions
Finds out number of data items to be transfered in a dataflow transaction
Figures out sequence order number for the given transaction
Setup different arrays for holding distinct objects
Finds out destination segment direction (left/right)
Setup best/optimal guard and enable values in ACC array for maximum performance

Figure 5.11: *SegBusEmulatorView* class operations' description.

ReadPSM() - This method is responsible for reading and parsing XML schema of the PSM model of the modeled solution (application + platform).

getProcessID() - This method extracts and returns the process ID from the passed string.

getNrMaster() - From the XML schema of the P-SDF model, this method allows to find out total number of *masters* which initiates data transfers.

getDataItems() - The method extracts and returns total number of data items to be transferred in a given dataflow transaction string between two processes. The number of data items do not depend on the package size at this stage.

getTSequence() - This method returns the transaction sequence number from a provided dataflow transaction string.

getCT() - This method takes out and return the CT value from a provided transaction string.

AddToThreadPool() - This method creates a thread pool using an instance of *ExecutorService* class from the *java.util.concurrent* package.

The size of the thread pool depends on the number of items in all lists that has been populated during parsing phase. Objects (in the form of items), from all lists, are added into the thread pool before emulation.

ArraySetup() - During the XML parsing process, the emulator generally establishes three distinct arrays by employing this method: an array for holding segments; an array for holding ***FU***s; and an array for holding all the masters. All these arrays are exhaustively used during emulation.

createACCArray() - This method is part of the ACG engine of the emulator application (discussed in Chapter 6). After the XML parsing has been finished for both P-SDF and PSM models, this method allows to create ACC array to be used during code generation after successful emulation. The details of the ACC array and code generation has been discussed briefly in Chapter 6.

getDestSegment() - As part of ACG, this method finds out the direction of inter-segment package transfer pertaining to involved dataflow transaction between two processes.

getMaxSequence() - This method returns the largest number from the transaction sequences in the P-SDF model.

setupGuardEnable() - This method setups best/optimal values of guard and enable fields in the ACC array to exploit maximum performance from as many as possible parallel transfers (discussed in Chapter 6).

isMoreTransaction() - This method determines about remaining transfers in the current transaction sequence. It returns *true* when there exists anymore transfers in the current sequence number, otherwise it returns *false* so that the parsing process moves further.

GenerateProgramLines() - This method is used to generate the control code for the arbiters.

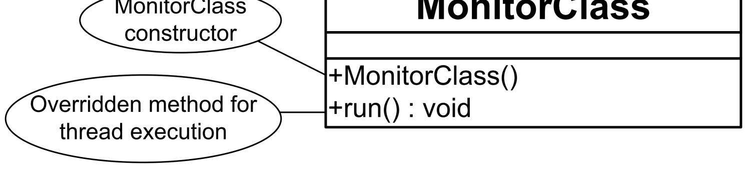


Figure 5.12: *MonitorClass* class operations' description.

Figure 5.12 depicts the structure of the *MonitorClass*. An object of this class acts as a thread during execution. This class is responsible for analyzing the status flags for all ***FU***s and monitors the activity inside other platform elements which are executing as threads during emulation. When the object of this class detects no communication activity within the platform, it sets particular flag to inform the emulator about the end of ongoing emulation. This class basically contains two methods: *MonitorClass()* - the constructor method of this class; and a *run()* method which executes when thread of this class is in the runnable state.

During the typical operation of the emulator, all threads execute concurrently and experience different thread states over the duration of emulation to depict intrinsic characteristics of hardware.

5.5 Emulation and Estimation

The final step of the emulation process is to emulate the platform configuration after initial setup. In general, application processes communicate with each other at different time instant after performing dedicated computation on the supplied data. The emulator extracts execution sequence from the P-SDF model and forwards them to relevant application processes in the initial setup. During emulator development, some timing factors are skipped that are less important in estimating performance. For instance, this study did not include the time necessary to synchronize between two adjacent clock domains, converging at the ***BU***s. This time is parametrized, but a value of two clock ticks is usually considered, at the translation of any signal across two clock domains. This study also did not compute the time necessary for the ***SA***s to set the grant signal for a particular request and corresponding master responds, due to a similarly low value, which is also overlapping in time with the ongoing activities within the segments.

When the designer supplies the XML schemes to the emulator, the tool parses the models, builds the communication matrix, instantiates the threads corresponding to platform elements (communication elements and application processes), supply particular value from communication matrix to each ***FU*** and starts the emulation process. Upon completion of the emulation, the emulator returns results from platform elements' execution. Some of the results are listed below:

- Total clock ticks (TCT) consumed for the operation of the ***CA*** and each of the ***SA***s.
- Total inter-segment requests received by ***CA*** and by each of the ***SA***s.
- TCT consumed by each of the ***BU***s.

- Start and end times of each ***FU***.
- Number of packages transferred to left and right directions from each ***BU***.

The clock tick's counter is incremented in ***SA*** and ***CA*** at numerous moments. Each ***SA*** has its own counter for counting clock ticks and the execution time for each device is computed separately (described in Chapter 7). For instance, the ***SA*** increments the clock tick's counter while checking the incoming requests from ***FU***s in the segment. It increments the counter when it receives intra- or inter-segment transfer request from one of the ***FU*** in the segment. If the request is for inter-segment transfer, it forwards the request to ***CA*** and increment the counter accordingly. While setting and resetting grant signal in response to any request, it also updates the clock tick's counter.

During the time limit for any transfer, the ***SA*** always increment the clock tick's counter continuously till the time limit ends. The ***CA*** increments the clock tick's counter every time when it checks for any incoming inter-segment transfer request from a ***SA***. It increments the counter while setting and resetting grant signal for any inter-segment transfer request. Furthermore, when one of the segment finishes its job in an inter-segment transfer, the ***CA*** resets the necessary signal associated with particular segment and increments the clock tick's counter.

5.6 Summary

This chapter introduced emulation technique for estimating performance aspects of desired *SegBus* configuration. The chapter described how the XML schemes can be generated from the models, specified in DSL, and introduced mechanism to emulate the modeled configuration in early stages of the development process.

The emulation-based solution enables us to analyze any platform configuration with respect to performance figures. Based on emulation results, it's the job of the designer to decide which particular configuration will be best suited for the final implementation. Such decisions in the early stages of design process not only improve the quality of eventual system in terms of performance, but also improves power consumption up to some extent [36]. The granularity level of application components can also be balanced in order to eliminate the traffic congestion located at certain ***BU***s, that will further improve the overall performance. Thus, the methodology allows a designer to adjust the high-level design in a way to take full benefits from the features exposed by the platform.

Chapter 6

The VHDL Snippets

This chapter presents a model-based approach for the generation of low-level control code for the arbiters, to support application implementation and scheduled execution on the *SegBus* platform. The approach considers model-based development (MBD) support as a key to model application at two dif-

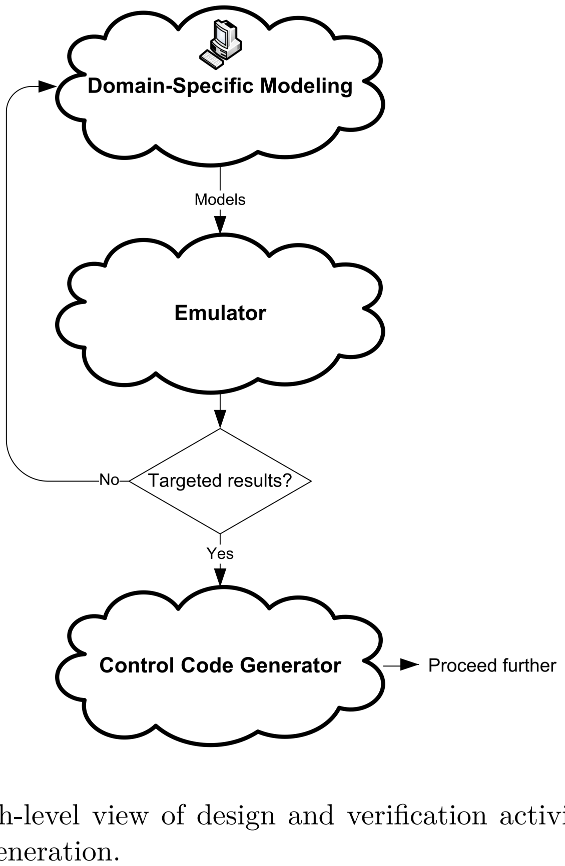


Figure 6.1: High-level view of design and verification activities leading to ultimate code generation.

ferent abstraction levels, namely as Package Synchronous Dataflow (P-SDF) and Platform Specific Model (PSM), using the SegBus DSL, as described in Chapter 4. Both models are transformed into Extensible Markup Language (XML) schemes, and then utilized by the emulator program (discussed in Chapter 5) to, further, generate the "application-dependent" VHDL code, the so-called "snippets". The obtained code is inserted into a specific section of the platform arbiters, which is treated as an execution schedule for the target application. Figure 6.1 depicts a high-level view of the design and verification activities employing the proposed framework finally leading to automatic control code generation (ACG).

6.1 Significance and Usage

Seceleanu et al. [41] introduced the definition of the low-level control code in the form of (manually obtained) VHDL snippets targeted for different arbiters of the *SegBus* platform. They introduced the control procedures for proper application execution over different segments of the platform. The proposed framework uses such definition as 'standard reference' and introduce certain methods in the emulator to generate application-dependent VHDL snippets in an automated fashion.

The activity of the elements of the platform is directed with the help of the local and central arbiters. They operate based on predefined policies, captured in the form of a "program". This is further implemented within the (VHDL described) body of the units as "VHDL snippets". The respective code must take into consideration both the local (within segments) and the global (platform level) structure and allocation, as well as the P-SDF of the application, trying to maximize the number of possible parallel operations. The results that are described here illustrate a procedure to automatically obtain the content of the snippets.

The control flow of both ***SA***s and of the ***CA*** is illustrated in Figure 6.2. During the operation of arbitration, each arbiter runs arbitration rounds indefinitely to control grant activities [20]. Briefly, in each round, the arbiter first checks if a master (in case of ***SA***) or a segment (in case of ***CA***) already has a bus grant for intended communication or not. In case there exists no previously approved grant, it checks any possible new requests for bus grant, and if there is any, it approves a single request at a time based on round-robin arbitration scheme and continue further for another arbitration round. Furthermore, in the other case when there exists a device or segment which has got the bus grant in the previous arbitration round, the arbiter sets appropriate signals in order to allow successful communication between the concerned devices or segments. Then, the arbiter monitors the amount of data transfers until it is finished, and finally it resets the grant and other involved signals and continue towards the next arbitration round.

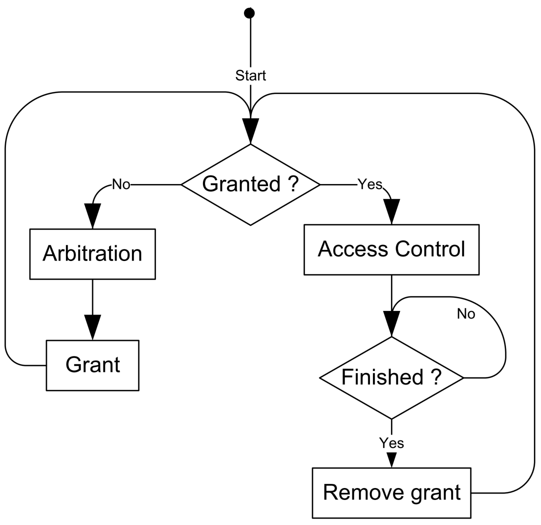


Figure 6.2: Arbiter control flow [41].

The ***SA***s and the ***CA*** are VHDL defined modules, with a similar structure [41]. The code implements the operational flow of Figure 6.2, running with multiple parameters as required by the platform specification. This research study sees the application as a set of correlated transactions that must be ordered in their execution by the arbiters. The specification of the schedule - as supplied by the P-SDF representation, is provided by a snippet introduced in the ***SA*** or the ***CA*** codes, representing the projection of the application flow at the respective level and location. The snippets correspond to the middle block - "Arbitration specification" in the arbiter structure of Figure 6.3.

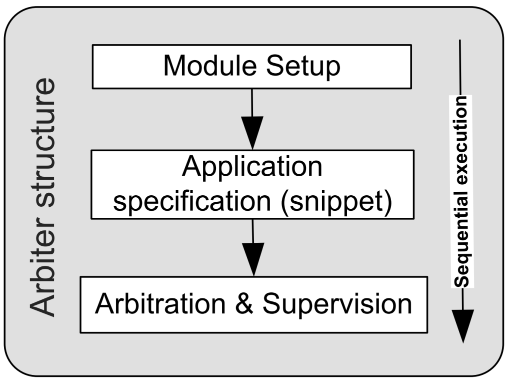

Arbiter structure
Module Setup
Application specification (snippet)
Arbitration & Supervision
Sequential execution

Figure 6.3: Arbiter code structure [41].

The block-level structure (see Figure 6.3), which is common in both central and segment-level arbiters [41]. The general tasks for arbitration in form of numerous procedures and functions are grouped into two different blocks, named as, "Module Setup" and "Arbitration & Supervision". The functionality of these two blocks in an arbiter are independent of any particular application. These blocks are also responsible for reading input; and producing output data to specific signals. The other important tasks include granting bus requests to requesting masters, counting the total number of transactions and resetting certain signals after each grant activity.

The middle block "Arbitration specification" is directly dependent on the target application and it is actually serves as the execution schedule for the application. Here, the research goal is to generate the control code from the emulator, in the form of VHDL snippets for arbiter's middle block in an automated manner. The execution schedule indeed is the scheduling of the grant decisions. In case of segment arbiter, it is the scheduling of different masters in a segment. While in case of central arbiter, it is the scheduling of the grant decisions for different segments involved in inter-segment transfers. Hence, the generated snippet is extremely vital for successful execution of an application on a particular platform instance.

The arbitration program elements. Being a part of a certain arbiter, when the snippet code is executed properly, it follows the exact sequence of actions as it was modeled in P-SDF. Thus, the snippet is a sequence - the *program* - of signal assignments - the *execution lines.*

```
...
-- A SA program line:
program(5) <= (guard => 0, source => 0, dest => 1,
   dest_seg => 0, togrant => 0, count=16, enables=13);
...
```

The *program* is thus a multi-dimensional vector consisting of a number of execution lines with several fields [41][42], described below. The other code blocks of the arbiter make use of different fields of this particular vector and may read or write them as needed. The signal assignment allows the implicit concurrency mechanism of VHDL to guide the selection of a program line, as available at a given execution moment.

In brief, the execution line fields can be described as follows.

program(x)

Basically, x can be seen as the *Program Counter*, and *program(x)* represents the x line of arbitration code. x also provides reference for accessibility from / to other lines of instructions.

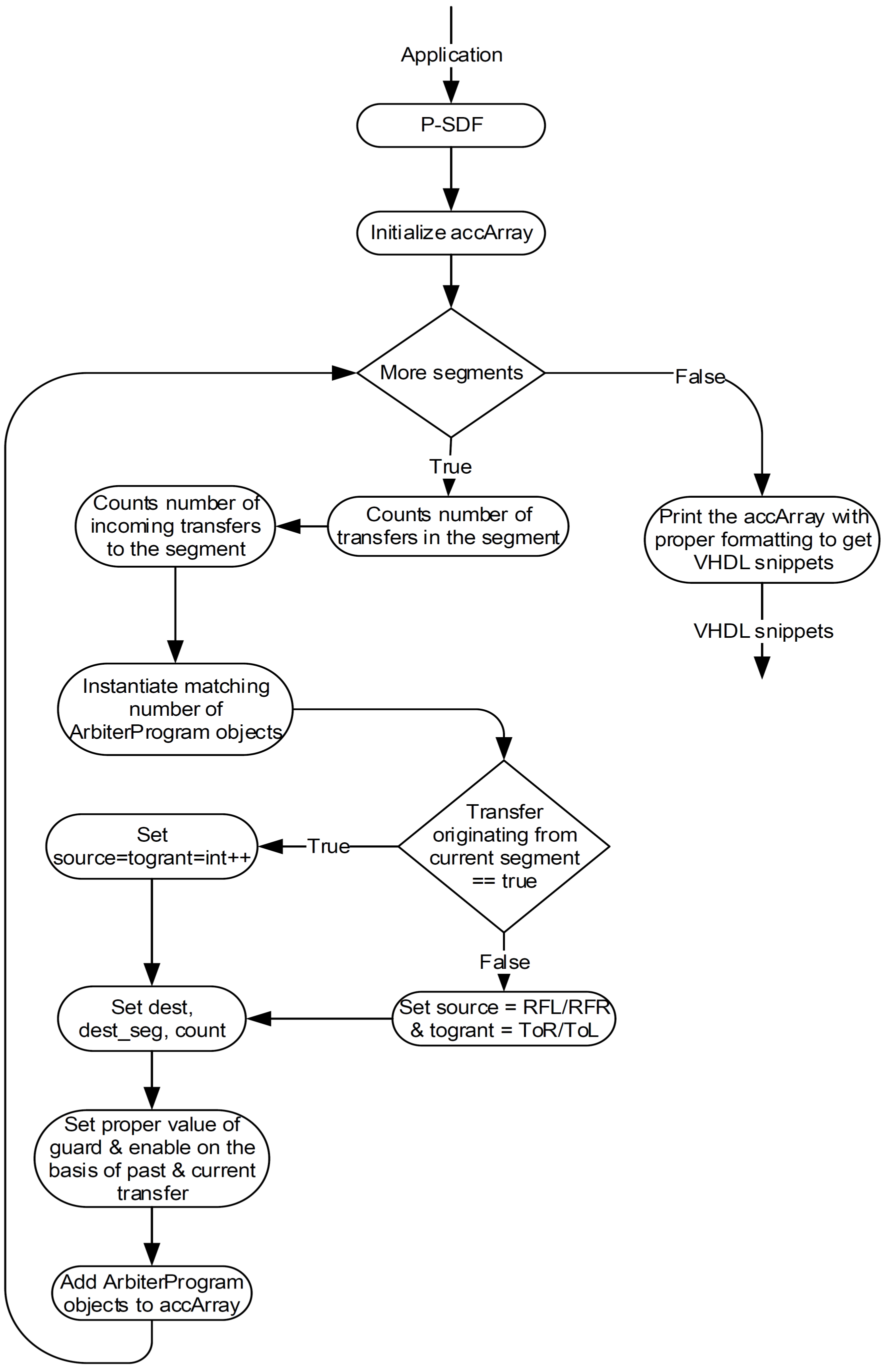

Application
P-SDF
Initialize accArray
More segments
False
True
Print the accArray with proper formatting to get VHDL snippets
VHDL snippets
Counts number of transfers in the segment
Counts number of incoming transfers to the segment
Instantiate matching number of ArbiterProgram objects
Transfer originating from current segment == true
True
Set source=togrant=int++
False
Set source = RFL/RFR & togrant = ToR/ToL
Set dest, dest_seg, count
Set proper value of guard & enable on the basis of past & current transfer
Add ArbiterProgram objects to accArray

Figure 6.4: Code extraction process for segments in the platform.

guard

When *guard* = 0, the respective line is *enabled*, that is, the arbiter may consider it for selection. When *guard* > 0, the line is disabled, that is, it cannot be considered in the arbitration. The arbiter marks a line as *executed* whenever the respective *count* value reaches 0, by establishing *guard = nrLines*, since *nrLines* is the total number of program lines in the *program* vector, associated with the given arbiter.

source

For ***SA*** case, this field contains the address of the requesting master - the initiator of a transfer request. Devices on the *SegBus* platform (masters, slaves) are identified by a unique number. For the ***CA***, this field contains the address of the initiating *segment*.

dest

The address of the targeted device - the slave.

dest_seg

The target slave's segment address.

toGrant

This is the instruction for the arbiter to grant the requesting master. At this moment the field is preserved for future developments.

count

This field identifies the number of packages the master has to send to the specified slave.

enables

This field points to a particular execution line in the program vector. When the execution line where this field contained in completes its execution, the ***SA*** in response *enables* that particular execution line (the targeted line in this context) which this field is pointing by subtracting 1 from targeted line's guard value so that the targeted line can start execution in the next round of arbitration. For the sake of clarity for readers, an execution line's *guard > 1* implies that this particular line is not enabled at the moment and the other previous and possibly dependent operations must have to be finished before this line starts execution.

6.2 Execution Schedule Generation

Figure 6.4 illustrates the general flow of the code generation process for segments after the parsing of the P-SDF model has been done (as discussed in Chapter 5.5) and the model resides now inside the emulator's internal variables and arrays. *ArbiterProgram* is a class in the emulator application which is basically a data structure with various data fields. It is used to represent a single line of the any specific arbiter's *program*. The code structure of this class is shown below.

```
public class ArbiterProgram
{
    public int program;
    public int guard;
    public int source;
    public int dest;
    public int dest_seg;
    public int togrant;
    public int count;
    public int enables;
    public int sequence;
    public boolean dirtyBit;
}
```

Firstly, the emulator analyzes number of originating and incoming transfers in each segment. On the basis of this information, it creates the respective number of *ArbiterProgram* objects. Secondly, it sets the *dest* field with the target process ID and *dest_seg* field with the segment ID where the target process is placed. If the transfer is originated from a master in the current segment, then it sets the *source* value of each object with an integer number in increasing order and *togrant* = *source*, otherwise the transfer is considered to be coming from a different segment via left/right ***BU***. In this case, the *togrant* = *ToR/ToL* and *source* = *RFL/RFR* are set according to the direction of the transfer. The *count* contains number of packages for this transfer (data items divided by the package size). The *program* field contains the order number of the execution line and the *sequence* field contains the relative order number of the execution line according to P-SDF model.

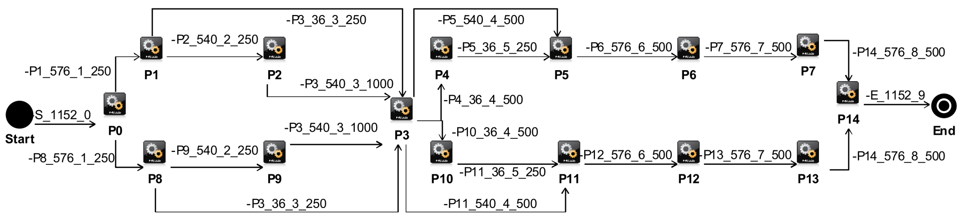


Figure 6.5: P-SDF model of an example application.

The *guard* and *enables* fields are important to introduce parallelism in the platform. An execution line is executed by the respective ***SA***, when its *guard* signal is zero. The emulator application sets the values of *guard* and *enables* field on the basis of ordering sequence of transfers. If two or more transfers occur at the same ordering sequence, it sets appropriate values to both fields so that parallel transfer can occur. For instance, the P-SDF model of the example application in Figure 6.5 contains two parallel transfers from process *P0* at sequence order 1. As per application requirements, both transfers needs to be completed in parallel before moving towards further transfers. The execution lines associated with these two transfers are given below:

```
program(0) <= (guard => 0, source => 0, dest => 1,
   dest_seg => 0, togrant => 0, count=16, enables=13);
program(1) <= (guard => 0, source => 1, dest => 8,
   dest_seg => 0, togrant => 1, count=16, enables=2);
```

A similar approach is taken with respect to the VHDL code to be generated for the ***CA*** operations. The only difference here is that the *source* and *dest* fields always refer to specific source and destination segments respectively for a particular data transfer rather than pointing to some *master/slave* devices in the same or different segment(s), as described earlier.

After performing successful emulation (as discussed in Chapter 5.5), and the obtained performance results are up to a desired level, the designer ultimately generates, with the help of automatic control code generation (ACG) engine, the application-dependent control code (in the form of synthesizable VHDL snippets) of the arbiters to be used in the final implementation. Following, an excerpt from the generated control code of an example application is shown. Note that the thesis uses the notations: ToR/ToL - the destination is the ***BU*** to the right / left of the current ***SA***); RFL - the request comes from left segment; and RFR - the request comes from the right segment.

```
--    VHDL Snippet for "Segment 1"
program(0) <= (guard => 0, source => 0, dest => 1,
   dest_seg => 0, togrant => 0, count=16, enables=13);
program(1) <= (guard => 0, source => 1, dest => 8,
   dest_seg => 0, togrant => 1, count=16, enables=2);
program(2) <= (guard => 1, source => 2, dest => 2,
   dest_seg => 0, togrant => 2, count=15, enables=3);
...
program(10) <= (guard => 1, source => 11, dest => 10,
   dest_seg => 0, togrant => 11, count=1, enables=11);
program(11) <= (guard => 1, source => 12, dest => 11,
   dest_seg => 1, togrant => ToR, count=15, enables=12);
program(12) <= (guard => 1, source => 8, dest => 11,
   dest_seg => 1, togrant => ToR, count=1, enables=0);
```

When the parsing process is done, the emulator creates:
• The *accCAArray*: a single-dimensional array, where each element in the array represents an execution line of the ***CA***.
• The *accArray*: a 2-dimensional array where each column represents an execution line of a ***SA***, while each row consists of execution lines associated with any particular ***SA***.

The above two arrays contain the key data of execution schedule for ***CA*** and ***SA***s, respectively. These arrays are employed to generate the VHDL snippets in well-formaed form as required by the related arbiter.

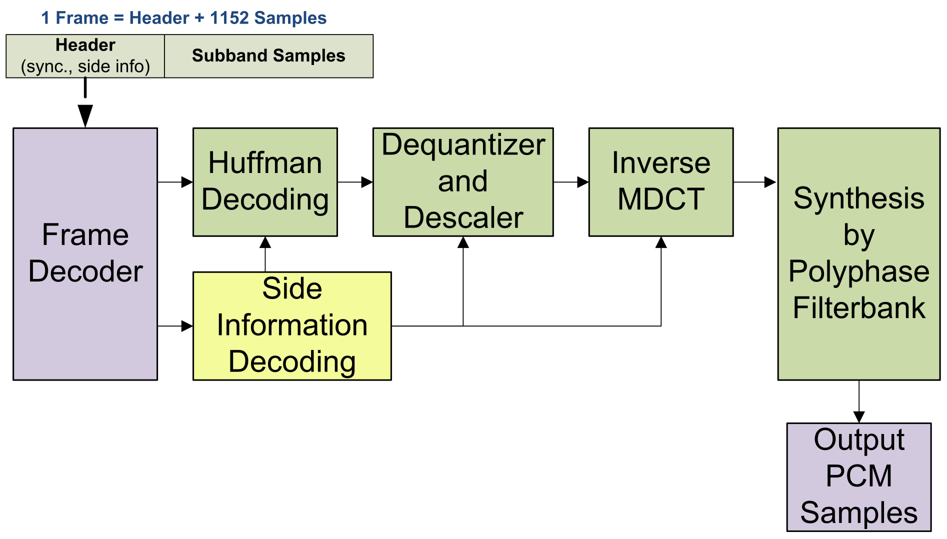
6.3 Summary

This chapter introduced MDA-based design methods to generate the application-dependent control code for a distributed platform, the *SegBus*. The chapter described methods to model application at P-SDF and PSM levels by employing *SegBus DSL* and run emulation using *emulator* program to get performance aspects of the modeled configuration. The emulator program has further evolved to generate the arbiters' low-level control code, in the form of VHDL *snippets*, which are then to be inserted in a specific block of (segment or central-level) arbiters as an execution schedule for any given application.

Chapter 7

The Overall Framework

This chapter formally employs and validates the proposed framework on a real example application. In this way, the chapter depicts the practicality and true effectiveness of the framework at each stage of the design methodology. Below, the chapter discusses briefly the important steps required to design and implement the chosen application on the *SegBus* platform making extensive employment of the proposed framework.



1 Frame = Header + 1152 Samples
Header (sync., side info)
Subband Samples
Frame Decoder
Huffman Decoding
Dequantizer and Descaler
Inverse MDCT
Synthesis by Polyphase Filterbank
Side Information Decoding
Output PCM Samples

Figure 7.1: Block diagram of the Layer III simplified MP3 decoder [17].

7.1 Example using the Framework

The thesis manifest the effectiveness of the proposed framework with an example of modeling, emulating and generating the execution schedule of a simplified stereo MP3 audio decoder application [17] on an instance of the *SegBus* platform.

Figure 7.1 illustrates the block diagram of the MP3 decoder. The first block of the decoder (frame decoder) performs (optional) error-checking and synchronization on the encoded bit-stream, which later logically divided into *frames*. Each frame consists of a header, side information and coded samples. After the decoding process, the decoder produces PCM samples which are used to reconstruct the audio data. Interested readers can further explore the detailed decoding process in [15].

7.1.1 Application Partitioning and Modeling

This research goal is started by the partitioning of the target application before modeling it in the DSL. During the partitioning process [50], the designer thoroughly analyzes the application based on prior experience, knowledge and the availability of IP components in the *SegBus* library. The designer researches the feasibility of running different application processes either as hardware modules (selected from the library) or as software modules. At this stage, the application processes are further decomposed down to the right granularity level on the basis of available library components. Similarly, certain software modules are also mapped on some hardware modules resulting in potential communication traffic remaining internal to that unit. In that case, the communication cost will be zero.

By employing the DSL, the designer then models the partitioned application of the MP3 decoder in (platform-independent) P-SDF form, as shown in Figure 7.2. In brief, process *P0* represents frame decoding, *P1/P8* - scaling on the left/right channel, *P2/P9* - dequantizing left/right channel, etc. The description about different attributes in each data transfer between processes has already been discussed in Chapter 5.3. Next, the communication matrix of the application is generated from the P-SDF model (see Figure 7.2) and is exposed in Figure 7.3. Each entity in the communication matrix shows the amount of data items (irrespective of package size) to be transferred from one particular process to another.

7.1.2 Configuring the Platform and Application Mapping

For PSM, the designer first forwards the communication matrix of the application to the (independent) *SBTool* [19] utility to obtain an optimum estimate of the allocation of the application processes over different platform configurations. Figure 7.5 illustrates the estimation of the possible

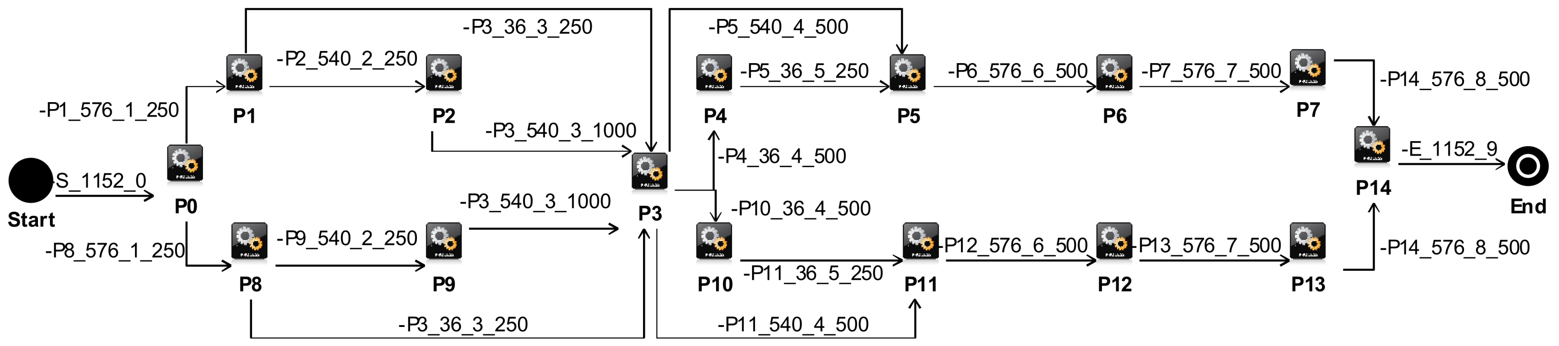


Figure 7.2: P-SDF model of the MP3 decoder.

To / From	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
P0	0	576	0	0	0	0	0	0	576	0	0	0	0	0	0
P1	0	0	540	36	0	0	0	0	0	0	0	0	0	0	0
P2	0	0	0	540	0	0	0	0	0	0	0	0	0	0	0
P3	0	0	0	0	36	540	0	0	0	0	36	540	0	0	0
P4	0	0	0	0	0	36	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	576	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	576	0	0	0	0	0	0	0
P7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	576
P8	0	0	0	36	0	0	0	0	0	540	0	0	0	0	0
P9	0	0	0	540	0	0	0	0	0	0	0	0	0	0	0
P10	0	0	0	0	0	0	0	0	0	0	0	36	0	0	0
P11	0	0	0	0	0	0	0	0	0	0	0	0	576	0	0
P12	0	0	0	0	0	0	0	0	0	0	0	0	0	576	0
P13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	576
P14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7.3: The communication matrix of the partitioned-application.

optimum allocation of application processes on three different platform configurations, where segment borders in each configuration are marked as '||'.

Here, the designer selects and models the "3 segment" platform configuration with linear topology in order to validate the viability and efficiency of the proposed framework. The package size is set to 36 data items (words) in each package. The designer specifies the configuration information of the platform in an instance of the *SegBusPlatform* class. For instance, for a three segments configuration, the respective attributes are: $NrSegs = 3$, $IsCircular = false$; the designer also sets other relevant information, such as address and data bus widths, etc. Finally the application processes are mapped from the P-SDF model onto the particular segments of the platform - Figure 7.4.

7.1.3 Model Validation

In order to check the structural correctness of the design, the designer executes the validation suite "SegBus Constraints" (see Chapter 4.3) as many times as needed. This can be executed any time, for instance after making a new change in the model, the designer simply clicks the menu in the MagicDraw UML: *Analyze* $\rightarrow$ *Validation* $\rightarrow$ *Validation.* Then, in the opened dialog, select *SegBus Constraints* and press the *Validate* button. Suppose that the validation is executed immediately after instantiation of the platform instance (*SegBusPlatform*), the modeling tool will generate an error message of "Missing Segment" (nr. 5 - see Chapter 4.3) and will highlight the platform instance because the designer didn't define any segment yet.

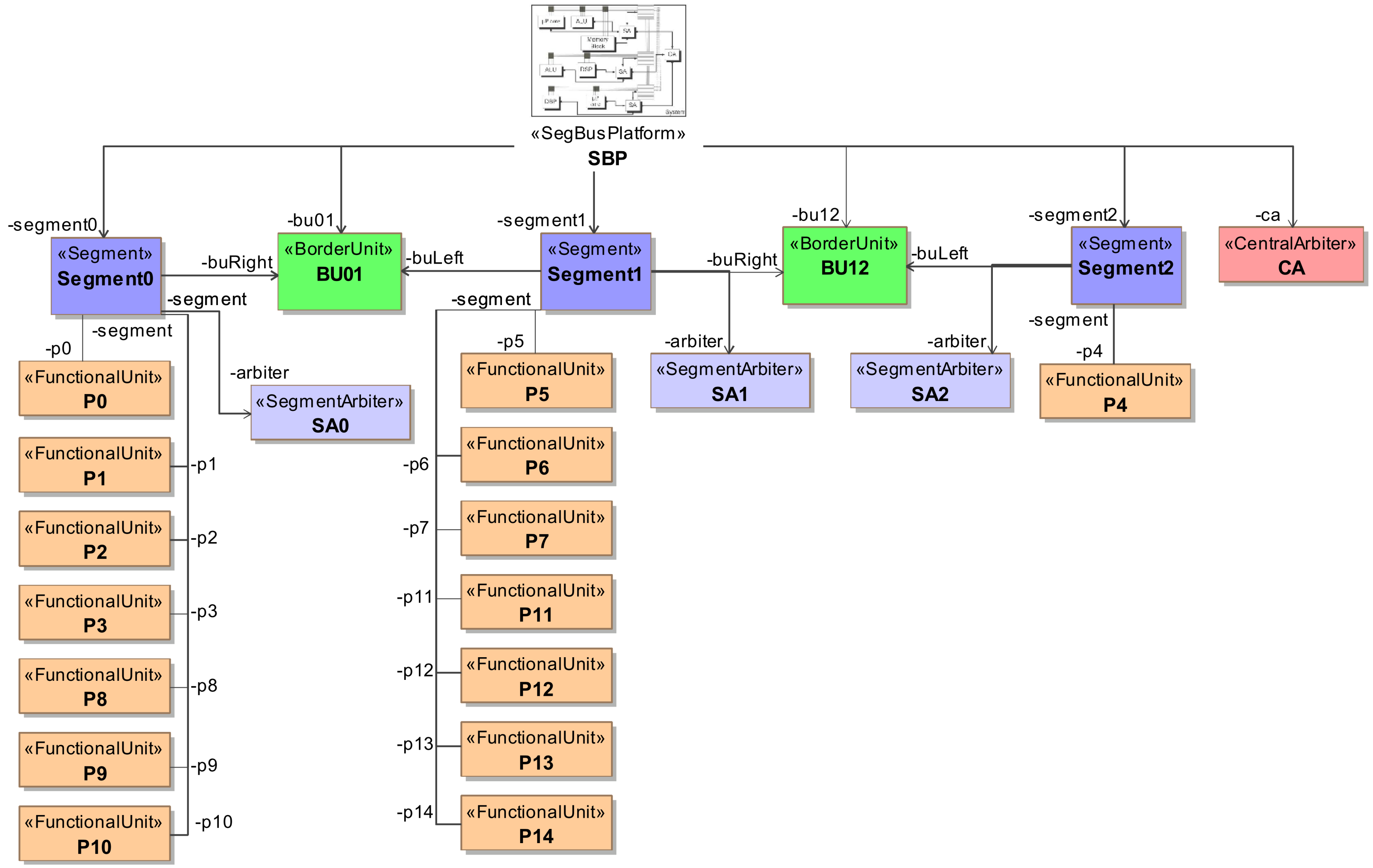
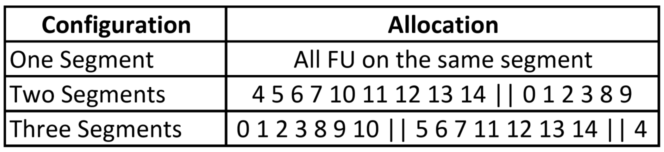


Figure 7.4: Example configuration of the *SegBus* platform with 3 segments and linear topology.



Configuration	Allocation
One Segment	All FU on the same segment
Two Segments	4 5 6 7 10 11 12 13 14 || 0 1 2 3 8 9
Three Segments	0 1 2 3 8 9 10 || 5 6 7 11 12 13 14 || 4

Figure 7.5: Allocation of processes on different platform configuration.

Moreover, if the designer models more/less number of segments unlike to what it was initially set in the 'NrSegs' attribute of the *SegBusPlatform* class. The validation process will produce three error messages (nr. 1, nr. 2 and nr. 4): the existing number of segments in the PSM model do not comply the 'NrSegs' attribute of the *SegBusPlatform* class; each of the segments does not contain any ***FU***; and the model does not have any ***BU*** yet. By taking the corresponding actions and running the validation again, these error messages will disappear.

Assume the designer wants to have a linear topology in a three segment configuration, then there must be two ***BU***s between the segments. On the other hand, if a circular topology is selected, then there must be three ***BU***s between the segments. If this requirement is not observed during modeling, the validation action will come up with an error message of "Number of ***BU***s"(nr. 3).

In another example, consider that the decision is to implement a platform with three segments. However, during the modeling phase, the designer introduces a fourth segment. This will conflict with the "Number of Segments" constraint, and the tool will provide the error message (nr. 1).

Finally, when the designer gets cleared from the validation process, the designer transforms the P-SDF and PSM models of the modeled system (application+platform) into XML schemes as it is briefly described in the next sub-section.

7.1.4 Model transformation of the example application

The designer first transforms the P-SDF model of the example application into an XML schema. Below the generated XML schema is shown after transforming the P-SDF model.

```
<?xml version='1.0' encoding='windows-1252'?>

<xs:schema targetNamespace="PSDF_MP3" xmlns="PSDF_MP3"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="End"><xs:sequence/></xs:complexType>
  <xs:complexType name="Start">
    <xs:sequence>
      <xs:element name="S_1152_0" type="P0"/>
    </xs:sequence>
  </xs:complexType>
```

```
<xs:complexType name="P14">
  <xs:sequence>
    <xs:element name="E_1152_9" type="End"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P5">
  <xs:sequence>
    <xs:element name="P6_576_6_500" type="P6"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P0">
  <xs:sequence>
    <xs:element name="P1_576_1_250" type="P1"/>
    <xs:element name="P8_576_1_250" type="P8"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P1">
  <xs:sequence>
    <xs:element name="P2_540_2_250" type="P2"/>
    <xs:element name="P3_36_3_250" type="P3"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P9">
  <xs:sequence>
    <xs:element name="P3_540_3_1000" type="P3"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P8">
  <xs:sequence>
    <xs:element name="P9_540_2_250" type="P9"/>
    <xs:element name="P3_36_3_250" type="P3"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P6">
  <xs:sequence>
    <xs:element name="P7_576_7_500" type="P7"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P7">
  <xs:sequence>
    <xs:element name="P14_576_8_500" type="P14"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P12">
  <xs:sequence>
    <xs:element name="P13_576_7_500" type="P13"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P2">
  <xs:sequence>
    <xs:element name="P3_540_3_1000" type="P3"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P10">
  <xs:sequence>
    <xs:element name="P11_36_5_250" type="P11"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="P4">
  <xs:sequence>
    <xs:element name="P5_36_5_250" type="P5"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="P13">
    <xs:sequence>
      <xs:element name="P14_576_8_500" type="P14"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="P3">
    <xs:sequence>
      <xs:element name="P4_36_4_500" type="P4"/>
      <xs:element name="P10_36_4_500" type="P10"/>
      <xs:element name="P11_540_4_500" type="P11"/>
      <xs:element name="P5_540_4_500" type="P5"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="P11">
    <xs:sequence>
      <xs:element name="P12_576_6_500" type="P12"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Similarly, the generated XML schema of the PSM model of the example application mapped on a 3-segments configuration with linear topology is shown below after model transformation.

```
<?xml version='1.0' encoding='windows-1252'?>

<xs:schema targetNamespace="segbusplatform" xmlns="segbusplatform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="P7">
    <xs:all>
      <xs:element name="segment" type="Segment1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="P4">
    <xs:all>
      <xs:element name="segment" type="Segment2"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="P5">
    <xs:all>
      <xs:element name="segment" type="Segment1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="P0">
    <xs:all>
      <xs:element name="segment" type="Segment0"/>
      <xs:element default="250" name="Time" type="xs:integer"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="SBP">
    <xs:all>
      <xs:element name="segment0" type="Segment0"/>
      <xs:element name="segment1" type="Segment1"/>
      <xs:element name="segment2" type="Segment2"/>
      <xs:element name="ca" type="CA"/>
      <xs:element name="bu01" type="BU01"/>
      <xs:element name="bu12" type="BU12"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="Segment0">
  <xs:all>
    <xs:element name="p0" type="P0"/>
    <xs:element name="p1" type="P1"/>
    <xs:element name="p2" type="P2"/>
    <xs:element name="p8" type="P8"/>
    <xs:element name="p9" type="P9"/>
    <xs:element name="p10" type="P10"/>
    <xs:element name="p3" type="P3"/>
    <xs:element name="arbiter" type="SA0"/>
    <xs:element name="buRight" type="BU01"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="P2">
  <xs:all>
    <xs:element name="segment" type="Segment0"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="P1">
  <xs:all>
    <xs:element name="segment" type="Segment0"/>
    <xs:element default="500" name="Time"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="CA"><xs:all/></xs:complexType>
<xs:complexType name="P11">
  <xs:all>
    <xs:element name="segment" type="Segment1"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="P10">
  <xs:all>
    <xs:element name="segment" type="Segment0"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="P12">
  <xs:all>
    <xs:element name="segment" type="Segment1"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="SA0"><xs:all/></xs:complexType>
<xs:complexType name="P8">
  <xs:all>
    <xs:element name="segment" type="Segment0"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="P3">
  <xs:all>
    <xs:element name="segment" type="Segment0"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="Segment2">
  <xs:all>
    <xs:element name="buLeft" type="BU12"/>
    <xs:element name="p4" type="P4"/>
    <xs:element name="arbiter" type="SA2"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="SA2"><xs:all/></xs:complexType>
```

```
  <xs:complexType name="Segment1">
    <xs:all>
      <xs:element name="buRight" type="BU12"/>
      <xs:element name="buLeft" type="BU01"/>
      <xs:element name="p5" type="P5"/>
      <xs:element name="p6" type="P6"/>
      <xs:element name="p7" type="P7"/>
      <xs:element name="p11" type="P11"/>
      <xs:element name="p12" type="P12"/>
      <xs:element name="p13" type="P13"/>
      <xs:element name="p14" type="P14"/>
      <xs:element name="arbiter" type="SA1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="P14">
    <xs:all>
      <xs:element name="segment" type="Segment1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="BU12"><xs:all/></xs:complexType>
  <xs:complexType name="P9">
    <xs:all>
      <xs:element name="segment" type="Segment0"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="P13">
    <xs:all>
      <xs:element name="segment" type="Segment1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="P6">
    <xs:all>
      <xs:element name="segment" type="Segment1"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="SA1"><xs:all/></xs:complexType>
  <xs:complexType name="BU01"><xs:all/></xs:complexType>
</xs:schema>
```

7.1.5 Execution with 3-Segments Configuration

The modeled configuration is emulated on *SegBus* emulator to analyze its performance. The emulation results of a 3 segments platform configuration are given below, where: 'CA' represents the central arbiter of the platform; 'Segment x' represents the segment and x denotes the ID (0,1,2,3,..); 'SAn' represents the segment arbiter associated with segment n; 'BUxy' represents the border unit between segment x and segment y; 'TCT' stands for total clock ticks consumed by related device. Clock frequency of segment 0, 1, 2 and central arbiter are set as 91MHz, 98MHz, 89MHz and 111MHz respectively. Here, it is important to observe that all segments are operating at different clock frequencies as this is an important hardware feature of the *SegBus* communication platform.

```
P0, Start Time = 10989ps, End Time = 75131793ps
P1, Start Time = 2758239ps, End Time = 137538324ps
P8, Start Time = 44010945ps, End Time = 141538320ps
P9, Start Time = 77087835ps, End Time = 194098707ps
P2, Start Time = 75131793ps, End Time = 196076727ps
P3, Start Time = 133648218ps, End Time = 270812916ps
P10, Start Time = 268823907ps, End Time = 272823903ps
P4, Start Time = 200466105ps, End Time = 297367980ps
P5, Start Time = 217161528ps, End Time = 342905420ps
P11, Start Time = 184253628ps, End Time = 344609488ps
P12, Start Time = 288426264ps, End Time = 400772304ps
P6, Start Time = 286609952ps, End Time = 402670248ps
P7, Start Time = 346354372ps, End Time = 458588168ps
P13, Start Time = 344548264ps, End Time = 460486112ps
P14 received last package at  461455492ps

CA TCT = 54475
Execution time = 490765275ps @ 111.00MHz

BU01:
                Total input packages = 32,
                Total output packages = 32
    Package Received from Segment 1 = 32,
            Package Transferred to Segment 1 = 0
    Package Received from Segment 2 = 0,
            Package Transferred to Segment 2 = 32
    TCT = 2336

BU12:
                Total input packages = 2,
                Total output packages = 2
    Package Received from Segment 2 = 1,
            Package Transferred to Segment 2 = 1
    Package Received from Segment 3 = 1,
            Package Transferred to Segment 3 = 1
    TCT = 146

Segment 0:
                Packages transferred to Left = 0,
                Packages transferred to Right = 32

Segment 1:
                Packages transferred to Left = 0,
                Packages transferred to Right = 0

Segment 2:
                Packages transferred to Left = 1,
                Packages transferred to Right = 0

SA0:        TCT = 34849,
                Total intra-segment requests = 124,
                Total inter-segment requests = 32
                Execution Time = 382955661ps @ 91.00MHz

SA1:        TCT = 46131,
                Total intra-segment requests = 137,
                Total inter-segment requests = 0
                Execution Time = 470720724ps @ 98.00MHz
```

```
SA2:        TCT = 35965,
                Total intra-segment requests = 0,
                Total inter-segment requests = 1
                Execution Time = 404066775ps @ 89.01MHz
```

7.1.6 Calculation of the Execution Time

The total execution time is calculated when all ***FU***s finish their tasks (setting the respective "Process Status Flag"), all packages are transmitted to their relevant destinations and the grant signals of all arbiters are *clear*.

Consider the total time consumed by ***SA***x (in this case, $x \in \{0, 1, 2\}$) to finish all the associated jobs as t_{SA_x}. the t_{SA_x} is calculated by multiplying the total clock ticks with the associated segment's clock period.

Then, the total execution time of the application is calculated by the taking the maximum of the total times consumed by the central arbiter and all segment arbiters that is *max* $(t_{SA_1}, t_{SA_2}, ..., t_{CA})$.

7.1.7 Emulation Results

Figure 7.6 shows the progress of each ***FU*** on a time line using a 3 segments, linear topology with package size of 36 data items. The figure shows the time at which any specific process finishes. For instance, process *P0* finishes the package transfers to process *P1* and *P8* at 75.13μs.

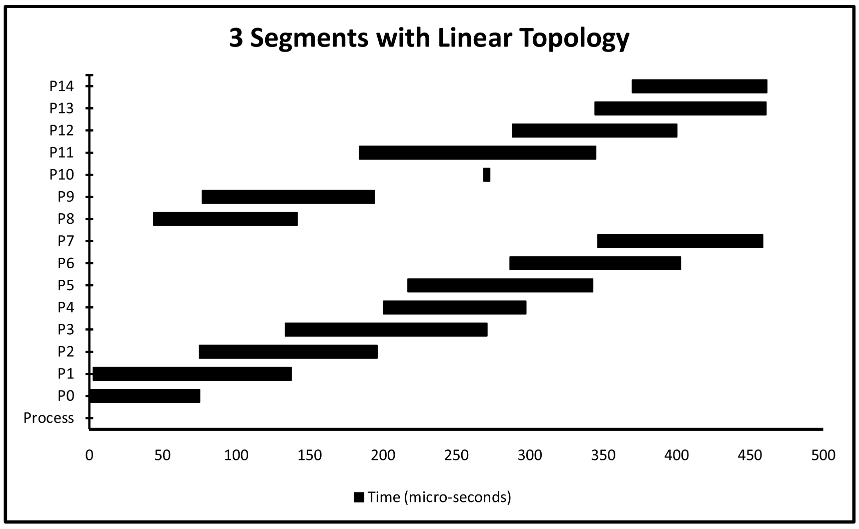


Figure 7.6: Progress on time of each application process in 3 segment, linear topology with package size of 36 data items configuration.

Computed as defined above, in the given configuration, the estimated total execution time for the application is 490.76μs. After running the same partitioned-application on the real platform instance in a identical platform configuration, an actual execution time obtained is 515.2μs. So, the emulated performance results suggest the emulator is 95% accurate.

Next, by keeping the same platform configuration, but the package size is changed to 18 data items (halve the payload). The result shows an estimated execution time of 560.16μs. The actual figure is 600.02μs, giving an emulator precision of around 93%.

Further, the platform configuration is then changed by shifting process *P9* from segment 0 to segment 2. The rest of the configuration kept stable, and the package size to 36 data items. The emulation estimated the execution time of this updated configuration as 540.4μs, while the actual execution time is 570.12μs, giving an emulator accuracy of just below 95%.

7.1.8 VHDL “Snippets” Generation

When the designer is satisfied after analyzing the emulation results. It’s time to move to the next step of the design methodology. Here the application-dependent control code of the platform’s arbiters are generated for a 3 segment configuration on the basis of the obtained performance levels after emulation. These generated codes for the arbiters are then inserted into a specific section of the related arbiters, as described in Chapter 6. Following, the generated control code, for all three segments for a package size of 36 data items, are shown.

```
--    VHDL Snippet for "Segment 0"
      ===========================
program(0) <= (guard => 0, source => 0, dest => 1,
   dest_seg => 0, togrant => 0, count=16, enables=13);
program(1) <= (guard => 0, source => 1, dest => 8,
   dest_seg => 0, togrant => 1, count=16, enables=2);
program(2) <= (guard => 1, source => 2, dest => 2,
   dest_seg => 0, togrant => 2, count=15, enables=3);
program(3) <= (guard => 1, source => 6, dest => 9,
   dest_seg => 0, togrant => 6, count=15, enables=4);
program(4) <= (guard => 1, source => 3, dest => 3,
   dest_seg => 0, togrant => 3, count=1, enables=5);
program(5) <= (guard => 1, source => 4, dest => 3,
   dest_seg => 0, togrant => 4, count=15, enables=6);
program(6) <= (guard => 1, source => 5, dest => 3,
   dest_seg => 0, togrant => 5, count=1, enables=7);
program(7) <= (guard => 1, source => 7, dest => 3,
   dest_seg => 0, togrant => 7, count=15, enables=8);
program(8) <= (guard => 1, source => 9, dest => 4,
   dest_seg => 2, togrant => ToR, count=1, enables=9);
program(9) <= (guard => 1, source => 10, dest => 5,
   dest_seg => 1, togrant => ToR, count=15, enables=10);
program(10) <= (guard => 1, source => 11, dest => 10,
   dest_seg => 0, togrant => 11, count=1, enables=11);
program(11) <= (guard => 1, source => 12, dest => 11,
   dest_seg => 1, togrant => ToR, count=15, enables=12);
program(12) <= (guard => 1, source => 8, dest => 11,
   dest_seg => 1, togrant => ToR, count=1, enables=0);
```

```
--    VHDL Snippet for "Segment 1"
      ===========================
program(0) <= (guard => 0, source => RFL, dest => 5,
   dest_seg => 1, togrant => RFL, count=15, enables=10);
program(1) <= (guard => 0, source => RFL, dest => 11,
   dest_seg => 1, togrant => RFL, count=15, enables=2);
program(2) <= (guard => 1, source => RFL, dest => 11,
   dest_seg => 1, togrant => RFL, count=1, enables=3);
program(3) <= (guard => 1, source => RFR, dest => 5,
   dest_seg => 1, togrant => RFR, count=1, enables=4);
program(4) <= (guard => 1, source => 13, dest => 6,
   dest_seg => 1, togrant => 13, count=16, enables=5);
program(5) <= (guard => 1, source => 16, dest => 12,
   dest_seg => 1, togrant => 16, count=16, enables=6);
program(6) <= (guard => 1, source => 14, dest => 7,
   dest_seg => 1, togrant => 14, count=16, enables=7);
program(7) <= (guard => 1, source => 17, dest => 13,
   dest_seg => 1, togrant => 17, count=16, enables=8);
program(8) <= (guard => 1, source => 15, dest => 14,
   dest_seg => 1, togrant => 15, count=16, enables=9);
program(9) <= (guard => 1, source => 18, dest => 14,
   dest_seg => 1, togrant => 18, count=16, enables=0);


--    VHDL Snippet for "Segment 2"
      ===========================
program(0) <= (guard => 0, source => RFL, dest => 4,
   dest_seg => 2, togrant => RFL, count=1, enables=1);
program(1) <= (guard => 1, source => 19, dest => 5,
   dest_seg => 1, togrant => ToL, count=1, enables=0);
```

Similarly, the generated control code for the ***CA*** is mentioned below.

```
program(0) <= (guard => 0, source => 0, dest_seg => 2,
   togrant => 0, count => 1, enables => 4);
program(1) <= (guard => 0, source => 0, dest_seg => 1,
   togrant => 0, count => 30, enables => 2);
program(2) <= (guard => 1, source => 0, dest_seg => 1,
   togrant => 0, count => 1, enables => 3);
program(3) <= (guard => 1, source => 2, dest_seg => 1,
   togrant => 2, count => 1, enables => 0);
```

7.2 Discussion

Based on these experiments, the accuracy of the emulator seems to be close to 95%. The errors are caused, mostly, by the imperfect modeling of the timing figures of the ***BU*** to ***SA*** control communication, the synchronization between clock domains, the granting activity of the ***SA***s, etc. [20].

However, firstly, these figures are very low (2 to 3 clock ticks), compared to the size of the package (36 data units). Secondly, most of these operations do overlap with each other, or with the data transfers. A clear identification of such events is impossible, hence one should accept the resulting errors. It becomes clear that, the higher the data package size, the less impact these figures should have in the estimation results from the emulator. This is due to the lower number of transfers, and hence, the inaccuracies of synchronization, granting, etc. actions of the ***SA***s.

Due to one of the considerations described in Chapter 5.1, the timing information illustrated in Figure 7.6 is not exact. This is due to the (variable) leading period of time during which each process awaits for data to be present at its input. However, as already mentioned, this does not have an impact on the overall application performance estimation, which, of course, includes such periods of time.

The tool helps the designer to estimate the communication bottlenecks expressed here as the time one package has to wait in one of the ***BU***s until it can be delivered to the next segment. The *useful period* (UP) of any given ***BU*** is expressed as the time (in clock ticks) required to load and then unload the data package, and it amounts to twice the size of a package. However, once a package is loaded, before unloading, the ***BU*** has to wait for a grant signal coming from the next segment - the *waiting period* (WP). As discussed and formalized in [20], WP is a non-deterministic value which may reach, at a maximum, the package size. An average value for WP ($\overline{WP}$) over the number of transfers executed by a certain ***BU*** can easily be computed given the data produced by the emulator (corresponding TCTs).

Considering the example at hand, for BU01 and BU12, the designer has the following values (clock ticks), respectively: UP01 = 2304, TCT01 = 2336, and $\overline{WP}$01 = 1; UP12 = 144, TCT12 = 146, $\overline{WP}$12 = 1.

Further, Figure 7.7 illustrates the activity graph, with respect to TCT, of 3 segments, linear topology configuration with different package sizes (18 and 36 data items).

7.3 Summary

This chapter validated the effectiveness of the proposed framework as a whole by employing it for modeling, mapping and implementing a real application. In the first part of the chapter, all the necessary steps of the framework required to develop optimum solution(s) are described for the targeted application to be executed on the *SegBus* platform. The chapter showed how the *SegBus* DSL is utilized to model application and platform in an easy and error-free manner.

The chapter also showed how the modeled system in DSL can be transformed into XML schemes which are later used by the emulator for performance estimation. This way, the models can be adjusted at this stage to achieve the desired performance from the actual to be implemented system.

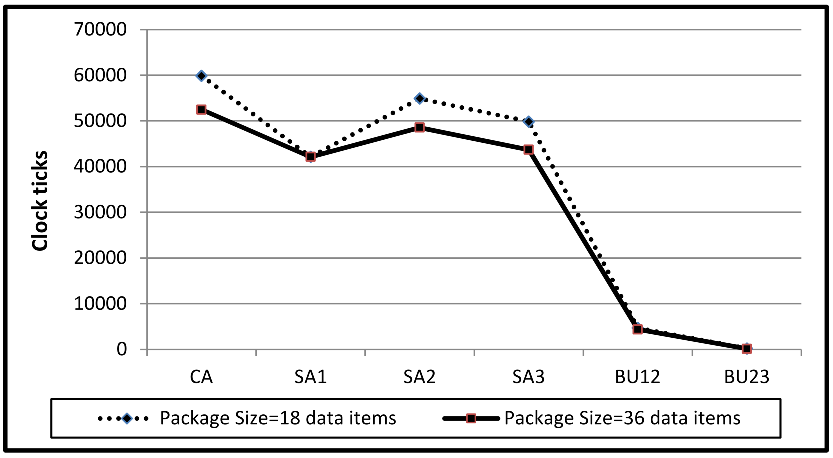


Figure 7.7: Activity graph of different platform elements in 3 Segments and linear topology configuration for 18 and 36 bit package sizes.

Furthermore, the chapter also described how the execution schedules for the platform's arbiters can be generated in an automated manner from the emulator as soon the desired system model is finalized.

In the second part of the chapter, the estimated accuracy of the emulation results is discussed. The chapter also discussed the possible reasons which cause the emulation results to slightly deviate. It has also been discussed that how to calculate various performance scores based on emulation results. This helps the designers to identify all performance barriers in the platform, and hence, directs them to make any necessary changes and fine tuning in the system models at high levels of abstraction such that maximum gains from MPSoC platform could be achieved in the final implementation.

Chapter 8

Conclusions

This concluding chapter of the thesis summarizes all the important contributions of the proposed framework, discussed in the previous chapters. This will give interested readers an overall view of the topics included in the thesis.

8.1 Thesis Contributions

The main contribution of this thesis is the introduction of a model-based development (MBD) and verification framework for a platform-based design approach, in the context of the distributed communications architecture known as the *SegBus* platform. The work is intended to provide automated procedures for platform build-up and application mapping in a correct and fast manner during early stages of the design process. The choice of MBD helps the designer to address Complexity of Design (CoD) - by the introduction of high-level design entry (HLDE); correctness of platform execution (CoPE) - with the support for structured design, and correctness related aspects; support for design reuse - with the help of library of components. Furthermore, it addresses the issue of designer expertise (DE) with the support of automatic code generation (ACG) as a hardware non-specialist is now able to develop applications not being fully aware of the specifics of the underlying hardware platform.

Within the proposed framework, in Chapter 4, the thesis firstly introduced a technique to set up a graphical interface (GI) in the form of a DSL based on a UML profile in an existing modeling tool. The GI supports modeling and mapping of a desired application on an instance of the *SegBus* platform. The thesis described in the form of graphical elements the principal structural elements of the platform and their structural relations with the help of related DSL customization classes. A library of reusable IP components was also introduced as part of DSL which allows the designer to include and refer to pre-built functional units in the modeling environment. The GI answers especially the CoD and CoPE challenges by providing the HLDE solution in the framework. In addition, by including the model validation features, the GI also serves the DE challenge.

The *SegBus* DSL provides an environment where a designer can model the platform and associate it with application components in a fast manner using different configurations. It helps designers to correctly model the application and platform and further helps in model transformation at later stages of development process. The validation suite embedded in DSL helps the designer rectify any problems in the models.

Secondly, in Chapter 5, the thesis also introduced an emulation technique targeting the *SegBus* platform for estimating performance of the chosen *SegBus* configuration. The thesis described how the XML schemes can be generated from the models, specified in DSL, and introduced mechanism to emulate the modeled configurations in early stages of the development process.

The emulation-based solution enables the designer to analyze any platform configuration with respect to its performance. Based on the emulation results, it's the job of the designer to decide which configuration is best suited for the final implementation. Such decisions in the early stages of design process not only improve the quality of final system in terms of performance, but also improves power consumption up to some extent [36]. The granularity level of application components can also be balanced in order to eliminate the traffic congestion observed in certain ***BU***s. This further improves the overall system performance. Thus, the methodology allows a designer to adjust the high-level design in a way to take full advantage from the features exposed by the platform.

Finally, in Chapter 6, the thesis discussed methods for an automated translation of the application models into "application dependent" control code based on VHDL. The generated code is then inserted in a specific block of (segment or central-level) arbiters as the execution schedule for a given application. This relieves the designer from the complex job of analyzing task sequencing in a parallel execution environment and data transfers between the platform sections. This technique addresses CoPE and DE based on ACG.

8.2 Future Directions

The current work is solely targeted to the *SegBus* platform but the model-based design principles could be applied to any MPSoC of choice. However, there is no "one size fits all" design methodology in MPSoC domain which is equally useful for every MPSoC. As future work, new tools have to be developed as the research work switches to another MPSoC platform hereby employing the same proposed principles. This could potentially lead towards uniformity in the design methods when targeting the MPSoC domain.

In the context of *SegBus* platform, current study addresses issues related to modeling of only one application at a time and its mapping onto an instance of the platform. The future work will also necessarily address modeling of more than one applications at a time and their mappings onto a single platform instance. The mapping and scheduling of the modeled applications will be an important challenge for the future work. In addition, extended support is expected to come in the form of arbiter code generation, for the implementation of the application schedules of more than one application.

Bibliography

[1] A. Jantsch, H. Tenhunen. *Networks on Chip.* Kluwer Academic Publishers, 2003.

[2] D. D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification.* Springer, 2009, ISBN: 978-1-4419-0503-1.

[3] R. Chen, M. Sgroi, L. Lavagno, A. S. Vincentelli, J. Rabaey. *UML for Real: Design of Embedded Real-Time Systems.* Kluwer Academic Publisher, Norwell, 2003.

[4] A. A. Jerraya, W. Wolf. *Multiprocessor System-on-Chips.* Morgan Kaufmann Publishers, 2005, ISBN: 0-12385-251-X

[5] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen and K. A. Houston. *Object-Oriented Analysis and Design with Applications.* 3^{rd} edition, Pearson Education, Inc., 2007, ISBN: 0-201-89551-X.

[6] D. Garlan and M. Shaw. *An Introduction to Software Architecture,* in *Advances in Software Engineering and Knowledge Engineering*, vol. 1, edited by V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jersey, 1993.

[7] C. Jaber.*High-level SoC Modeling and Performance Estimation: Application To A Multi-core Implementation Of LTE EnodeB Physical Layer.* LAP Lambert Academic Publishing, ISBN-13: 978-3659210891, 2012.

[8] M. Fowler and K. Scott. *UML Distilled. Second Edition* Addison-Wesley, ISBN: 020165783X, 2002.

[9] H. M. Deitel and Deitel & Associates, Inc., P. J. *JAVA How to Program, 7^{th} Edition.* Prentice Hall, ISBN: 9780136085676, 2006.

[10] G. Booch. *Object-Oriented Analysis and Design With Applications.* Addison-Wesley, 3^{rd} edition, 2007, ISBN-13: 978-0201895513.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design with UML.* Prentice Hall, 2nd edition, 2004.

[12] I. Jacobson, M. Christerson, P. Jonsson and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Professional, 1992.

[13] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process* Addison-Wesley Professional, 1999.

[14] B. Evjen, K. Sharkey, T. Thangarathinam, M. Kay, A. Vernet, S. Ferguson. *Professional XML* Wiley Publishing, Inc., ISBN: 978-0-471-77777-9, 2007.

[15] J. J. Thiagarajan and A. Spanias. *Analysis of the MPEG-1 Layer III (MP3) Algorithm Using MATLAB* Synthesis Lectures on Algorithms and Software in Engineering, 2011, doi:10.2200/S00382ED1V01Y201110ASE009.

[16] K. Lahiri, A. Raghunathan, S. Dey. *Design Space Exploration for Optimizing On-Chip Communication Architectures.* IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, vol. 23, no. 6, June 2004, pp. 952-961.

[17] C. Park, J. Jang and S. Ha. *Extended Synchronous Dataflow for Efficient DSP System Prototyping.* Journal of Design Automation for Embedded Systems, Springer Netherlands, vol. 6, no. 3, 2002, pp. 295-322.

[18] A. Molina and O. Cadenas. *Functional verification: approaches and challenges.* Journal of Latin American Applied Research, vol. 37, no. 1, 2007, pp. 65-69.

[19] T. Seceleanu, V. Leppänen, O. Nevalainen. *Improving the Performance of Bus Platforms by Means of Segmentation and Optimized Resource Allocation.* The EURASIP Journal on Embedded Systems, Volume 2009 (2009), Article ID 867362, doi:10.1155/2009/867362.

[20] T. Seceleanu. *The SegBus Platform - Architecture and Communication Mechanisms.* Journal of Systems Architecture, Vol. 53, Issue 4, April 2007, pp. 151-169. Doi:10.1016/j.sysarc.2006.07.002

[21] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. Hämäläinen, J. Riihimäki, K. Kuusilinna. *UML-Based Multiprocessor SoC Design Framework.* ACM Transactions on Embedded Computing Systems, vol. 5, no. 2, May 2006, pp. 281-320.

[22] A. Gamatié et al. *A Model-Driven Design Framework for Massively Parallel Embedded Systems.* ACM Transactions on Embedded Computing Systems (TECS), vol. 10, nr. 4, article no. 39, November 2011.

[23] S. Gérard, P. Feiler, J. F. Rolland, M. Filali, M. O. Reiser, D. Delanote, Y. Berbers, L. Pautet, I. Perseil. *UML&AADL '2007 Grand Challenges.* ACM Special Interest Group on Embedded Systems, vol. 4, nr. 4, October 2007, pp. 1-17.

[24] N. Genko, D. Atienza, G. D. Micheli, L. Benini. *Feature-NOC emulation: a tool and design flow for MPSoC.* IEEE Circuits and Systems Magazine, vol. 7, 2007, pp. 42-51.

[25] A. Sandiovanni-Vincentelli, G. Martin. *Platform-Based Design and Software Design Methodology for Embedded Systems.* IEEE Design and Test of Computers, vol. 18, no. 6, Nov./Dec. 2001, pp. 23-33.

[26] M. Vidmantas, E. Kazanavičius. *Conception of a Multi-Platform System Software and Firmware Development Tool.* Periodical of Information Sciences, Issue 50, 2009, Vilnius University Publishing House, pp. 194-199.

[27] *International Technology Roadmap for Semiconductors.* 2007 Edition.

[28] T. Seceleanu, H. Tenhunen et al. *Multicore Processing and ARTEMIS.* Networking session at the IST 2006 Conference, Helsinki, Finland.

[29] B. Kienhuis, E. Deprettere, K. Vissers and P. van der Wolf. *An Approach for Quantitative Analysis of Application Specific Dataflow Architectures.* In Application-specific Systems, Architectures and Processors (ASAP), July 1997.

[30] B. Kienhuis, E. F. Deprettere, P. van der Wolf and K. Vissers. *A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach.* Lecture Notes in Computer Science, vol. 2268, Springer 2002, pp. 18-37.

[31] R. Arora, M. Mernik, P. Bangalore, S. Roychoudhury, S. Mukkai. *A Domain-Specific Language for Application-Level Checkpointing.* The International Conference on Distributed Computing and Internet Technologies, 2008, pp. 26-38.

[32] C. Consel, H. Hamdi, L. Réveillére, L. Singaravelu, H. Yu, C. Pu. *Spidle: a DSL approach to specifying streaming applications.* The 2^{nd} International Conference on Generative programming and component engineering, 2003, pp. 1-17.

[33] A. Ferrari, A. Sangiovanni-Vincentelli. *System design: Traditional concepts and new paradigms.* The IEEE International Conference on Computer Design: VLSI in Computer and Processors, 1999, pp. 2Ű12.

[34] A. S. Vincentelli and J. Cohn. *Platform-Based Design and Software Design Methodology for Embedded Systems.* IEEE Design and Test of Computers, vol. 18, no. 6, 2001, pp. 23-33.

[35] A. Koudri, J. Champeau, D.Ãulagnier, P. Soulard. *MoPCoM/MARTE Process Applied to a Cognitive Radio System Design and Analysis.* The 5th European Conference on Model Driven Architecture - Foundations and Applications, 2009, pp. 277-288.

[36] T. Seceleanu et al. *Application Development Flow for On-Chip Distributed Architectures.* In Proceedings of the 21st IEEE International System-on-Chip Conference (SOCC), 2008, pp. 163 - 168.

[37] E. A. Lee and D. G. Messerschmitt. *Synchronous Dataflow.* Proceedings of the IEEE, vol. 75, no. 9, Sep. 1987, pp. 1235 - 1245.

[38] T. Lindroth, R. Lavinia, T. Seceleanu, N. Avessta, J. Teuhola. *Building a UML Profile for On-chip Distributed Platforms.* The 30th International Computer Software and Applications Conference (COMPSAC), 2006, pp. 372-373.

[39] P. Liu, C. Xiang, X. Wang, B. Xia, Y. Liu, W. Wang and Q. Yao. *A NoC Emulation/Verification Framework.* In Proceedings of 6th International Conference on Information Technology: New Generations, 2009, pp. 859-864.

[40] W. Risi, P. López, D. Marcos. *HyCom: A Domain Specific Language for Hypermedia Application Development.* The 34th Annual Hawaii International Conference on System Sciences (HICSS-34), Vol. 9, 2001, pp. 163-168.

[41] T. Seceleanu, I. Crnkovic, C. Seceleanu. *Transaction Level Control for Application Execution on the SegBus Platform.* The 33th IEEE Computer Software and Application Conference (COMPSAC), 2009, pp. 537-542.

[42] T. Seceleanu et al. *MultiCast protocol for SegBus platform.* In the proceedings of NORCHIP, 2009, pp. 1-6.

[43] R. Silaghi, A. Strohmeier. *Integrating CBSE, SoC, MDA, and AOP in a Software Development Method.* In proceedings of 7th International Enterprise Distributed Object Computing Conference (EDOC), 2003, pp. 136.

[44] M. Thompson, T. Stefanov, H. Nikolov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere. *A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs.* In proceedings of 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007, pp. 9-14.

[45] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, E. Deprettere. *Daedalus: Toward composable multimedia MP-SoC design.* In proceedings of 45th ACM/IEEE Design Automation Conference (DAC), 2008, pp. 574-579.

[46] M. Waseem, L. Apvrille, R. Ameur-Boulifa, S. Coudert and R. Pacalet. *Abstract Application Modeling for System Design Space Exploration.* In proceedings of 9th EUROMICRO Conference on Digital System Design (DSD'06), 2006, pp. 331-337.

[47] M. Waseem, L. Apvrille, R. Ameur-Boulifa, S. Coudert and R. Pacalet. *A UML-based Environment for System Design Space Exploration.* In proceedings of 13th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2006, pp. 1272-1275.

[48] S. Demathieu, F. Thomas, C. André, S. Gérard, F. Terrier. *First Experiments using the UML Profile for MARTE.* The 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), May 2008, pp. 50-57.

[49] R. Thomson, S. Moyers, D. Mulvaney, V. Chouliaras. *The UML-based Design of a Hardware H.264/MPEG-4 AVC Video Decompression Core.* The 5th International UML for SOC Design Workshop, June 2008.

[50] D. Truscan, T. Seceleanu, J. Lilius, H. Tenhunen. *A Model-based Design Process for the SegBus Distributed Architecture.* In Proceedings of the 15th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), 2008, pp. 307-316.

[51] E. S. Shin et al. *Round-robin Arbiter Design and Generation.* In Proceedings of the 15th International Symposium on System Synthesis, 2002, pp.243-248.

[52] H. Kopetz. *The Complexity Challenge in Embedded System Design.* In 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008, pp. 3-12.

[53] E. Riccobene, A. Rosti, P. Scandurra. *Improving SoC Design Flow by means of MDA and UML Profiles.* In 3rd Workshop in Software Model Engineering (WiSME), 2004.

[54] G. Schelle, D. Grunwald. *Onchip Interconnect Exploration for Multicore Processors utilizing FPGAs.* 2nd Workshop on Architecture Research using FPGA Platforms, 2006.

[55] Apache Ant$^{\text{TM}}$.
`http://ant.apache.org/`

[56] MagicDraw Open API user guide, version 17.0.
`http://www.magicdraw.com/`

[57] Java Programming Language.
`http://java.sun.com/`

[58] Object Management Group.
`http://www.omg.org/`

[59] *Eclipse Modeling - Model-to-Text Transformation.*
`http://www.eclipse.org/modeling/m2t/`

[60] MagicDraw UML.
`http://www.magicdraw.com/`

[61] Model-Driven Architecture.
`http://www.omg.org/mda/`

[62] Matlab - The Language of Technical Computing.
`http://www.mathworks.com/`

[63] SCADE Suite.
`http://www.esterel-technologies.com/products/scade-suite/`

[64] OMG. *Object Constraint Language (OCL) 2.0 Revised Submission, version 1.6.* Jan. 2003.

[65] *Unified Modeling Language Specification, version 2.2, 2009.*
`http://www.omg.org/spec/UML/2.2/`

[66] *MetaObject Facility, version 2.0, 2006.*
`http://www.omg.org/spec/MOF/2.0/`

[67] *XML Metadata Interchange, version 2.1.1, 2007.*
`http://www.omg.org/spec/XMI/2.1.1/`

[68] *Common Warehouse Metamodel, version 1.0, 2001.*
Available at `http://www.omg.org/`

[69] *The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems.*
`http://www.omgmarte.org/`

[70] *Architecture Analysis & Design Language.*
`http://www.sae.org/`

[71] *POSIX Threads.*
`http://standards.ieee.org/findstds/interps/1003-1c-95_int/`

[72] *OpenMP API.*
`http://openmp.org/wp/`

[73] *Message Passing Interface (MPI).*
`http://www.mcs.anl.gov/research/projects/mpi`

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture

ISBN 978-952-12-3028-8
ISSN 1239-1883

Moazzam Fareed Niazi

A Model-Based Development and Verification Framework for Distributed SoC Architecture