Université de Montréal

ESys.Net
A New .Net Based System-Level Design Environment

par

James Lapalme

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Maîtrise présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Décembre, 2003

Université de Montréal
Faculté des études supérieures
Ce mémoire intitulé :

ESys.Net
A New .Net based System-Level Design Environment

présenté par :

James Lapalme

a été évalué par un jury composé des personnes suivantes :

Dr. Houari Sahraoui
président-rapporteur

Dr. Jean-Pierre David
directeur de recherche

Dr. El Mostapha Aboulhamid
codirecteur

Dr. Gabriela Nicolescu
membre du jury

Mémoire accepté le 23 mars 2004

Sommaire

Avec l'arrivée des systèmes embarqués qui incorporent un nombre croissant de composantes logicielles, il devient plus critique que jamais de rétrécir l'écart qui existe entre la modélisation de niveau système et l'implantation.

Dans ce travail nous illustrons, par le développement d'un nouvel environnement de modélisation et simulation basé sur le langage de programmation C#, le potentiel insoupçonné de modélisation matérielle/logicielle du .Net Framework, potentiel qui a permis de pousser SystemC au-delà de ses limites.

Notre environnement, appelé ESys.Net, utilise les concepts de programmation avancés de .Net et C# tels que la réflectivité, la programmation par attribut et la création dynamique de délégués, afin de créer un environnement plus polyvalent que SystemC.

ESys.Net a pu bénéficier de plusieurs constructions puissantes du génie logiciel en raison de l'utilisation de .Net comme base. Les résultats expérimentaux que nous avons recueillis démontrent que ces constructions n'entraînent pas de pénalités d'exécution significatives.

Mots clés : modélisation, .Net, C#, simulation, système-sur-puce, langages de description, SystemC

Abstract

The need to bridge the gap between system level and implementation level modeling is becoming critical as embedded systems incorporate more software components. Through this work we illustrate, by developing a new modeling and simulation environment, how we can use the .Net Framework through the use of the C# programming language to model hardware/software systems and alleviate the different shortcomings of C++ that are hindering the evolution of SystemC.

Our environment, called ESys.Net, uses the advance programming features of .Net and C# such as reflection, attribute programming and dynamic delegate creation to produce a flexible solution that is meant to be an evolution of SystemC.

By using .Net as a basis for ESys.Net, we have inherited many powerful software engineering constructs. The experimental results that we have gathered demonstrate that these constructs do not incur a significant performance penalty.

Keywords: modelling, simulation, .Net, C#, description languages, system-on-chip, SystemC

Table of Contents

Figure List

Table List

Code Example List

Abbreviation List

CAD	Computer Assisted Design
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLS	Common Language Specification
CTS	Common Type System
ECMA	European Computer Manufacturers Association
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
FIFO	First In, First Out
FPGA	Field-Programmable Gate Arrays
HDL	Hardware Description Language
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
ISO	International Organization for Standardization
JVM	Java Virtual Machine
NOC	Network On Chip
OOP	Object-Oriented Programming
OSCI	Open SystemC Initiative
PCB	Printed Circuit Board
RTL	Register Transfer Level
SCV	SystemC Verification Library
SDL	System Description Language
STOC	SpecC Technology Open Consortium
UCI	University of California Irvin
VES	Virtual Execution system
VHDL	Very High Speed Integration circuit Hardware Description Language
XML	Extensible Markup Language

I would like to dedicate this to my mother and father
who have always pushed me to better myself,
and to Zachary and Marie-Josée, the two loves of my life,
without whom this would have no meaning

Acknowledgements

Above all, I would like to thank El Mostapha Aboulhamid, the initiator of the ESys.Net project, without whom there would be no ESys.Net. Thank you for believing in me and being very patient, your wisdom was invaluable.

I would like to thank Jean-Pierre David for his help and support.

I wish to express my gratitude to Luc for helping me with the design and for putting up with my endless babbling.

Also, I would like to thank my Mom, Bruce, Irene, Dan, Marie-Josée, Steven and Gabriela for their important contributions.

I especially would like to thank Marie-Josée, my soul mate, for her constant support and patience. I would not have made it without you.

James Lapalme

Preface

The need to bridge the gap between system level modeling and implementation modeling is becoming pressing as embedded systems incorporate more software components. Maybe a change of paradigm is needed for hardware and system modeling?

Most current hardware description languages have two significant advantages over generic programming language: syntactic brevity for hardware semantics and better constructs for model verification. However even these advantages are melting away with the emergence of languages like Asml. Even SystemC, a C++ based solution, has been able to incorporate fairly simple syntactic constructs -through the use of macro- to provide hardware semantics. Assertion based verification capabilities have usually only been supported by hardware description languages and specialized verification languages but generic programming languages are beginning to incorporate those capabilities such as Eiffel and Asml.

The major limiting factor of using generic programming languages for hardware modeling is that hardware semantics are not directly present. But what if we could add the missing metadata? What could we do if we had the power of certain high level programming languages: the reflective capabilities of Java, the polymorphism of Ruby, the elegance of ML, or the simple power of Perl? Would all of this change the way we think of system modeling and hardware modeling? The .Net Framework currently makes interoperability between languages almost seamless. It also permits the integration of custom metadata.

This thesis presents a new environment called Esys.Net that we have created for system-level modeling and simulation. We developed this solution in response to our

frustrations with SystemC, frustrations caused by (i) the complexities of SystemC's underlying implementation language (C++) (ii) its overly complex library (iii) its complex design that makes custom modifications verify difficult (iv) and especially its lake of third-party tool integration capabilities. When we first started developing ESys.Net, our main objective was simply to port SystemC to the .Net Framework in order to eliminate certain of SystemC's downfalls. However, as the project advanced, we rapidly discovered the full potential of the .Net Framework and the C# programming and decided to re-engineer SystemC in order to take full advantage of the underlying technologies. We added many new features to the overall design of the new environment in order to create a solution meant to be an evolution of SystemC.

Introduction

For many years now, the ever growing gap between the available computing power offered by hardware platforms and that used by the software applications running on these platforms has been tolerated because of the need for platform independent software, independence required because of the difference in life expectancy between hardware and software products. Today, with the emergence of embedded systems, it is imperative that these new systems take full advantage of the computing power available on the underlying hardware platform and that a perfect balance may be reached between software and hardware.

A major hurdle limiting the production of better systems, especially embedded ones, is the existence of an annual 30% gap between the growth of chip complexity and human design productivity [27] . To overcome this design crisis, it is clear that sophisticated CAD tools and new design methodologies are necessary to help designers model, simulate, partition and verify complex hardware/software systems. Over the past several years, many researchers have looked towards the creation of better design environments integrating powerful tools for system modeling, simulation, partitioning and verification [44] .

1.1 HDLs and SDLs

Before Moore's law [48] pushed the elaboration of hardware systems by a single individual out of the realm of reality, systems were developed in an almost artistic way by electronic engineers. When systems became overly complex, tools were created to help teams communicate various aspects of a hardware design [47] . The first tools available were called Hardware Description Languages (HDLs) and were a spin-off of programming languages. A good overview of an HDL is [67] :

> "In electronics, a **hardware description language** or **HDL** is a standard text-based format for describing either the behaviour or the structure, or both, of an electronic circuit. Most HDLs are restricted to describing digital circuits, but there are exceptions. HDLs have two purposes. First, they are used to write a model for the expected behaviour of a circuit before that circuit is designed and built. The model is fed into a computer program, called a simulator, that allows the designer to verify that his solution behaves correctly. Second, they are used to write a detailed description of a circuit that is fed into another computer program called a logic compiler. The output of the compiler is used to configure a programmable logic device that has the desired function. Often, the HDL code that has been simulated in the first step is re-used and compiled in the second step."

The basic difference between an HDL and a traditional imperative programming language is the presence of a certain number of modeling semantics:

- Parallel processing elements (e.g. process)
- Timing constraint elements (e.g. clock, time)
- Structural decomposition elements (e.g. modules)
- Interconnection elements (e.g. signals)
- Ports

There exists no formal document that describes the modeling semantics of an HDL but current examples support the above modeling elements even though there syntax or name may differ. Figure 1 illustrates some of the above concepts on a circuit schematic.

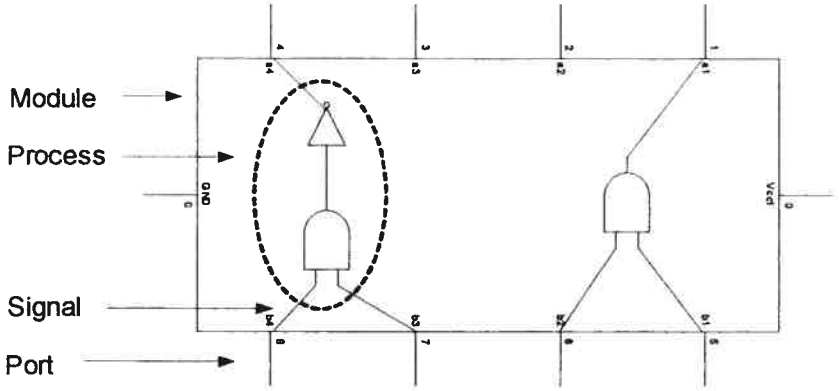

Figure 1 : Simple circuit

Because of the advances in electronic component interconnections, the concept of HDLs has been extended in recent years with the semantics of communication channels that permit the modeling and abstraction of complex communication mediums. These new tools are called System-Level Description Languages (SDLs) [59] .

The industry and academics have for several years tried to create better HDLs and SDLs to aid with the never ending design crisis. One method that has been explored for the creation of these tools (HDLs and SDLs) is a library-based approach which consists of taking an existing programming language and adding to it the missing constructs and semantics for hardware and system design [7] . A second approach is a standalone one consisting of the simple creation of a new language.

Despite all these efforts, system designers still need new modeling and simulation solutions. This is mainly due to a mandatory set of requirements for an efficient modeling and simulation framework which are still not provided by a single existing environment:

- Easier software components specification and their integration into an overall Hardware/software system specification [66] ;
- Clean programming features to enable less error-prone models, easier specification for complex systems and reuse of such specification for further designs [62] ;

- Introspection features for easier debugging and analysis of complex systems [36] [22] .
- Possibility of annotating models for different purposes, e.g. directing synthesis or hooking to verification tools, creating user friendly HDL syntax [50] ;
- Translation to a standard intermediate format to enable the design of CAD tools independently of the used description languages [40] ;
- Integration to distributed web-based design environment and easy system documentation to facilitate cooperation between different design groups and to allow remote processing [14] ;
- Multi-platform and multi-language features for describing and designing the overall embedded systems composed of heterogeneous components [34] ;
- Easier memory management to accelerate the specification process and to eliminate an important source of errors [54] .

1.2 Specific Goals

The SystemC [65] modeling environment has been gaining momentum for the past several years and has become a de facto standard for systems-level modeling. However, because its underlying implementation is based on C++, its evolution is rapidly slowing down. We believe that SystemC will have grave difficulties in keeping up with new environments, which will incorporate many advance system-level modeling features for operating system and hardware/software system modeling [44] [4] [3] . The existence of an environment such as SystemC is very important because it is one of the few good modeling and simulation environments that is "free" and "open source". Most current environments are products developed and sold by big corporation that are demanding high licensing fees.

The objective of this thesis is to propose an environment for system-level design that (1) provides most of the concepts present in high-level modeling and simulation solutions, (2) respects all the requirements enumerated above and (3) preserves comparative performances with existing environments, by using C# and the .Net Framework.

The mission of our environment is to use the proven environment of SystemC as a basis for a new solution that will also be "free" and "open source". Our environment brings to the hardware/software modeling community a new solution that has all the benefits of SystemC without having most of its drawbacks. We hope that our solution will offer designers a good alternative to expensive proprietary solutions.

1.3 Outline of this Document

Chapter 2 gives an overview of the different environments available for the modeling and simulation of hardware/software systems. It presents a brief introduction of the new challenges facing system designers today and in the future. It also presents current software frameworks that might help in solving these new problems.

Chapter 3 presents the .Net Framework and the C# programming languages. It then goes on to explain the advanced programming features that these two technologies support which have permitted us to create a new system-level design environment called ESys.Net.

Chapter 4 highlights the various elements that make up the ESys.Net framework. They are presented individually, their semantics explained and their uses illustrated. Many code examples are given to help the reader understand the subtleties of the environment.

Chapter 5 is entirely dedicated to our simulation kernel, since the major design differences between SystemC and ESys.Net are in the simulation kernel. This chapter gives descriptions and compares the design of both environments.

Chapter 6 discusses the advantages and disadvantages of the ESys.Net environment; some experimental results are presented also.

Finally, chapter 7 summarizes the project and suggests directions for further research.

Chapter 2 State of the Art

The complexity of reality surpasses greatly our capability of synthesis and analysis. To cope with this inadequacy, we simplify things in order to create models that we can manipulate and understand. Hardware system design and the new area of hardware/software system codesign are domains that we definitely cannot cope with without simplification and abstraction. These domains are plagued by an ever growing complexity fed by technological advancements and new consumer needs.

To simplify and model these complex systems, hardware description languages have been used for about 40 years now [12] [6] . They became widely used with the adoption of VHDL as an IEEE standard in 1987. There are two classes of HDLs. Standalone HDLs have their own syntax, compilers and analyzers, whereas HDLs that are in fact libraries are based on existing programming languages such as C++, C or Java. Each approach has pros and cons as we will illustrate in the following sections. We will show how recent developments in the software domain, by the introduction of new frameworks, can help in the domain of hardware/software co-design. These languages permit the description of systems in a clear and standard way, permitting the easy exchange of information between people.

2.1 Standalone Languages

This first class of HDLs is composed of languages that were developed from scratch for the sole purpose of hardware and hardware/software systems modeling. The vast majority of these were developed by the industry for the industry.

2.1.1 VHDL[31]

The development of VHDL was initiated in 1981 by the United States Department of Defence to address the hardware life cycle crisis [16] . VHDL was meant (i) to provide a unified notation for describing electronic systems at various levels of abstraction, (ii) to be both machine and human readable, (iii) to support the communication of design data, (iv) to aid the maintenance, modification and procurement of hardware, and finally (v) to support the development, verification, synthesis and testing of hardware designs through a tool agnostic description. VHDL was a purely hardware description language. Early on designers noticed the absence of software and system primitives such as FIFOs, and system synchronization mechanisms such as semaphores, locks and shared variables, as well as object oriented paradigms which would help with design reuse. Also, VHDL was verbose and not well adapted to describe components which are lower than the gate level. Gate libraries were generally described using in-house scripts. Test benches were also generated using special scripts even though VHDL had many useful constructs to describe test benches. In fact, in some aspects, it is much richer than Verilog or more recent languages such as SystemC in developing gate level and register transfer level (RTL) test benches. Except for basic assertion based verification, VHDL did not provide any other verification capabilities. Some features were very specific to VHDL, such as the possibility of having metadata-like attributes. Metadata is very useful for tools that interpret a model either for simulation, verification, test coverage, test generation, or synthesis. It also has the notion of the separation between the interface of a component (described by an entity) and its functionality which is described by one or more architectures. This separation is unique to VHDL; the community had to wait for recent object-oriented hardware description languages to find similar capabilities. This separation between the interface and the behaviour was very useful in design space exploration and to some extent in design reuse. Another unique feature of VHDL is the concept of resolution functions which allow very well defined protocols to modify a signal by concurrent processes; this allowed the description of very high level modeling of interaction between subsystems. In total,

fifteen different IEEE standards around VHDL have been adopted such as VHDL-AMS [30] for analog design.

2.1.2 Verilog

Verilog was the main competitor of VHDL until the announcement of SystemC in 1999. Even though it appeared in 1985 it became an IEEE standard only in 1995[32] . It is a less verbose language than VHDL but comes with some limitations such as a narrow data type set, a resolution function restricted to "wired or" and "wired and" and no separation between interfaces and behaviour. However, it has better performances and a well defined foreign interface to hook to other languages. Currently, VHDL and Verilog are converging more and more in capability [4] .

2.1.3 SystemVerilog [54]

SystemVerilog, which was adopted as a standard by Accellera in June 2002, is an extension of Verilog. It can be seen as a stack of components aimed at verification, design and system modeling. For verification, it provides facilities both for test bench generation as well as assertions. For the design aspect, it provides many enhancements to Verilog, such as provision for communication interfaces and an enriched data type set similar to the C programming language. Since SystemVerilog is a new product and very few case studies have been published, simulation performance remains to be seen. Many companies donated different technologies to this environment. The white paper by S. Bailey [3] provides an excellent comparison between VHDL, Verilog and SystemVerilog. In that report we note that features available in VHDL but not in Verilog or vice-versa have been added to SystemVerilog, such as named events, partially strong typing, records and structures, hierarchy, reactive processes, interface abstraction, assertions and foreign interfaces, and system level primitives and mechanisms such as mailboxes, semaphores, dynamic process creation, etc. Some capabilities of VHDL have been omitted or only partially implemented such as operator overloading, general resolution functions, full-fledged attributes, configurations and binding.

2.2 Programming Language-based HDLs

The second class of HDLs is based on an existing language such as C++, C or Java. Existing programming languages are usually missing basic hardware description semantics such as concurrent behaviour, timing elements, communications elements etc.; so this second class of HDLs is usually implemented by either providing a framework which adds the necessary missing hardware semantics to the base programming languages or extends the languages with additional syntactic and semantic constructs – a superset approach. These are, with some exceptions, open source environments and commercial tools that are either less evolved or targeted to a specific niche; however, they are very useful in an academic environment. In the following sections we will describe the characteristics of some illustrative examples.

2.2.1 Handel-C and OCAPI [7]

Some articles have illustrated the advantages of using HDLs based on existing programming languages [8]. OCAPI is based on C++ and is very efficient at system level exploration while Handel-C is C-based and can generate efficient designs translating them to EDIF or VHDL for implementation on FPGAs. The strength of both environments and their seamless integrations can provide a very strong design flow from system level to FPGA implementation.

2.2.2 JHDL [5] [28]

JHDL is an object-oriented environment; it uses exclusively object-oriented constructs of Java for RTL hardware modeling, simulation and efficient implementation on FPGAs. The environment permits the description of synchronous digital logic circuit components and connections such as: static cells, Boolean gates, registers, "parameterizable" modules etc. JHDL was developed as an exploratory attempt to identify the key features and functionalities that a good FPGA tool needs. It has been also recently used for Intellectual Property blocks (IP) delivery through the internet.

2.2.3 SpecC [59]

SpecC was developed by the University of California, Irvine (UCI), and first appeared in 1997. In 1999, the SpecC Technology Open Consortium (STOC) was founded. As a result, the SpecC language was refined and extended, leading to its second generation, SpecC 2.0 which was approved by the STOC in December 2002. The SpecC language is a superset of the ANSI-C programming language. It is a formal notation intended for the specification and design of digital embedded systems. SpecC extends C with concepts essential for embedded systems design such as: behavioural and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling and timing. SpecC is one of the few existing environments that supporting explicit behavioural hierarchies. It focuses on an IP co-design methodology for modeling and design at the system level. SystemC channel and communication abstractions were inspired from the pioneer work done in SpecC.

2.2.4 SystemC [65]

SystemC, announced in September 1999 by OSCI (The Open SystemC Initiative), was met with much enthusiasm by both industry and academia. It was the first open source library approach environment based on C++. It is currently very popular for hardware-software system-level design. It provides all the basic concepts used by HDLs (e.g. modules, ports, signals, timing, etc.) and more abstract concepts (interfaces, communication channels, events, etc.). However, most of the features for software modeling are still missing in SystemC: dynamic process creation, process control (suspend, resume, kill, etc.), pre-emption, software specific communication primitives (monitors, semaphore, etc.). Many companies used SystemC to model in a very efficient way the system aspects of their design. As illustrated in a survey done by Doulos [17] the use of SystemC is mainly performance modeling, architecture exploration, and transaction level modeling and hardware-software co-simulation. The survey also shows that (i) standard HDLs continue to be used for hardware modeling and synthesis, (ii) a minority of users use SystemC for RTL synthesis and (iii) many companies are interested in operating systems and software scheduling

which are not currently available in SystemC 2.0. Given that SystemC 2.0 is in fact a C++ library it lends itself to the development of very sophisticated test benches and as shown in [13], SystemC surpasses largely specialized languages such as the E language or Vera which are meant for verification and test bench development. SystemC was a good catalyst for new contributions, i.e. transaction level modeling to deal with the increased complexity of models, software engineering methodologies for interoperability design reuse [10] , simulation of network topologies [43] , and functional verification [23] . Finally, it is the first open source HDL for which a language reference manual has been completed in order to submit it for an IEEE standard approval. In 2003 the OSCI announced the SystemC Verification Library (SCV) which added some verification and introspection capabilities to the existing environment [18] .

2.3 New Challenges in Modeling and Design

Needs have evolved from simply describing hardware at the RTL level to including communicating subsystems, abstracting communication and buses, and dealing with low power and interconnects.

Hardware/software systems are becoming a reality and complexity is increasing at an exponential rate. This rapid growth has pushed many to use third-party IP. Using high quality third-party IPs permits the reduction of design time while improving overall quality and facilitating the design of heterogeneous systems. However, IP reuse also brings different challenges such as:

- Finding effective ways of delivering IPs to customers
- Insuring a sufficient visibility of the IP so that customers may validate custom models and simulate the complete system containing in-house and IP components
- Providing the above features while protecting the intellectual property of the vendor.

Another important revolution in the domain of integrated circuits is the integration of non-microelectronic elements on a chip such as micro-optical and micro-mechanical

components. These complex and heterogeneous systems also produce different problems to solve at the modeling and simulation level. We should be able to model and simulate systems expressed using different languages, paradigms, and concerns as well as components described at different levels of abstraction and protection. If we examine the recent modeling and design environments illustrated by SystemC and SystemVerilog, we notice that the concerns are: performance of simulation, ease of programming and debugging. There are also software issues such as operating systems primitives, multithreading, provision for foreign interfaces and increased levels of abstraction, as well as verification needs for the integration of components and the interaction between them. Other concerns which are increasing in importance are power dissipation and the problems related to the shrinking technology going into the nanotechnology.

Through the long history of HDLs we notice the influence of parallel and simulation languages developed in the software domain on the development of HDLs. Verilog was a simile combination of an earlier HDL and Occam parallel-processing language. VHDL was also largely influenced by ADA and Verilog by C syntax. We think we should go a step further to where the hardware modeling will become only one aspect within a general software framework environment. The large success of existing HDLs was the possibility to go from an RTL description to a gate level implementation in a very efficient way. Up until now there have been no convincing success stories accomplished at higher levels of abstraction. Behavioural synthesis is still not adopted by the industry, and commercial tools at that level of abstraction have not been very successful. One success story may be the current transaction level modeling enabled by the introduction of SystemC. The focus seems to be on the combination of modeling and verification on one hand, and development reuse, delivery and integrations of third party IPs.

2.4 Recent Software Frameworks

Even the most recent modeling and design environments such as SystemC and SystemVerilog have many shortcomings. These are monolingual environments with limited capabilities of accessing models written in other languages. The error prone

programming and the lack of type-safe features in C++ hamper the development of SystemC. SystemVerilog will not be an open source environment which will be a hurdle to universities in developing and experimenting with CAD/EDA frameworks.

The .NET framework was announced in 2000, one year after the introduction of SystemC. In our opinion, it contains features that would have greatly influenced the language choice for implementing SystemC. Java was not chosen due to its lack of performance compared to C++, the absence of operator overloading and generic classes, etc. If we look at the C# programming language [19] introduced with .NET, we note that all these shortcomings have been removed. An excellent comparison of C#, C++ and Java is given in [2] . It shows how C# takes advantage of the strengths of Java and C++ and blends them in a very powerful and elegant language. The performance of C# is also confirmed by different publications [42] . Many features planned for implementation in SystemC or SystemVerilog are already implemented in an efficient way in C#, such as automatic garbage collection, safe pointers, software multithreading, mailboxes, semaphores, monitors, etc.

In contrast to the JAVA environment and its virtual machine aimed only at JAVA, .NET is a multilingual environment [39] [26] , a necessity in the domain of hardware/software modeling and design. It could be very beneficial to explore the capabilities of recent frameworks such as .NET in modeling, verifying and designing hardware/software systems. These frameworks bring many features to be adapted for hardware/software modeling and design such as: safe simulation of models, including models created by an unknown or semi-trusted third party, a consistent object-oriented programming environment whether the model is local or remote [39] [56] , increased reuse and multilingual support, and the existence of a published intermediate format that renders lower level tools independent of higher level modeling languages. As we can see, all these features can be applied to development and delivery of IPs, the modeling and simulation of heterogeneous systems as well as the development or integration of modeling, synthesis and verification tools.

We should also benefit from the characteristics of recent software frameworks that are nonexistent in SystemC or SystemVerilog, such as the ability to document a

model using metadata [50] , which can be accessed by reflection either to specify the simulation verification or synthesis semantics, the use of reflection to explore a model for verification, test coverage or refinement, and self-contained documentation using standards such as XML.

Chapter 3 Advanced Programming Features with C# and the .Net Framework

With time HDLs are beginning to integrate many features that we have come to expect of high level programming languages making them much more similar to software programming languages than hardware modeling languages. By using new software development tools and leveraging advanced programming features, improved library-approach HDLs and SDLs can be developed. This section presents two new software development tools, the .Net Framework and the C# programming language. The advanced programming features, supported by these two technologies, which have had the greatest impact on the development of ESys.Net, are also presented.

3.1 The .NET Framework

Virtual machines, intermediate languages and language independent execution platforms are not new. They were present with UNCOL in the 1950's to the JVM in the 1990's. Researchers have been fascinated with these concepts because they permit an alternative path to native compilers that have several benefits [45] :

- Portability: To implement n languages on m platforms, only $n + m$ translators are needed instead of n * m translators.
- Compactness: Source code is usually much more compact when translated to an intermediate format.
- Efficiency: By delaying the commitment to a specific native platform as much as possible, we can make optimal use of the knowledge of the underlying machine, or even adapt to the dynamic behaviour of the program.

- Security: High-level intermediate code is more amenable to deployment and runtime enforcement of security and typing constraints than low level binaries
- Interoperability: By sharing a common type system and high-level execution environment, interoperability between different languages becomes easier than binary interoperability. Easy interoperability is a prerequisite for multi-language library design and software component reuse.
- Flexibility: Combining high-level intermediate code with metadata enables the construction of (type safe) meta-programming concepts such as reflection, dynamic code generation etc.

The .NET core represented by the CLI (Common Language Infrastructure) is a new virtual machine execution platform which was standardized in December 2001 by ECMA and in April 2003 by ISO [20] .

3.1.1 General Presentation of .NET Framework [49]

The .NET Framework is a new platform that simplifies component-based application development for the highly distributed Internet environment. What sets the .NET framework apart from its rivals (such as the Java platform) is that its core, the CLI, was designed from the ground up to be a multi-language environment [26] [45] . At the center of the Common Language Infrastructure (CLI) is a unified type system, the Common Type System (CTS), and a Common Intermediate Language (CIL), which supports high-level notions (e.g. classes) and which is platform and programming language independent. The CTS establishes a framework enabling cross-language integration, type safety, and high performance code execution.

The CLI has four main components:

The Common Type System. The Common Type System (CTS) provides a rich type system that supports the types and operations found in many programming language families. It is intended to support the complete implementation of a wide range of programming languages

Metadata. The CLI uses metadata to describe and reference the types defined by the Common Type System. Metadata is stored ("persisted") in a way that is independent

of any particular programming language. Thus, metadata provides a common interchange mechanism for use between tools that manipulate programs (compilers, debuggers, etc.). Metadata is also used to augment the CIL representation of a source code.

The Common Language Specification. The Common Language Specification (CLS) is an agreement between language designers and framework (class library) designers. It specifies a subset of the CTS Type System and a set of usage conventions. Languages provide their users the greatest ability to access frameworks by implementing at least those parts of the CTS that are part of the CLS. Similarly, frameworks will be most widely used if their publicly exposed aspects (classes, interfaces, methods, fields, etc.) use only types that are part of the CLS and adhere to the CLS conventions.

The Virtual Execution System. The Virtual Execution System (VES) implements and enforces the CTS model. The VES is responsible for loading and running programs written in CIL. It provides the services needed to execute managed code and data, i.e. automatic memory management (Garbage Collection), thread management, metadata management etc. The VES also manages the connection at runtime of separately generated code modules through the use of metadata (late binding).

The CLI also gives the specification number of class libraries providing important functionalities such as thread interaction and reflection. It also provides XML [21] data manipulation, text management, collection functionality, web connectivity, etc.

Alongside the CLI core, .NET Framework presents a set of classes that add supplementary features such as web services, native and web forms, transaction, scalability and remote services, etc.

3.2 The C# Language

The C# language is a simple, modern, general-purpose object-oriented programming language that has become an ECMA and ISO standard [19] . It was intended for

developing software components suitable for deployment in distributed environments. Although most C# implementation (Microsoft , Xixiam , DotGNU [46] [15] [55]) used the CLI standard for its library and runtime support, other implementations of C# need not, provided they support an alternate way of getting at the minimum CLI features required by this C# standard.

In order to give the optimum blend of simplicity, expressiveness, and performance, C# supports many software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection [2] .

C# is intended for writing applications for both hosted and embedded systems ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions. Although C# applications are intended to be economical with regards to memory and processing power requirements, the language was not intended to compete directly on performance and size with C or assembly language.

3.3 Advanced Programming Features

Since the C# language relies on a runtime with the CLI's features; it inherits interesting characteristics such as a unified type system, thread and synchronization support, and automatic memory management just to name a few. It is sometimes hard to separate the C# language and the CLI because they are quite symbiotic so .Net/C# or CLI/C# will sometimes be used throughout this document.

There are three advanced programming features that .Net/C# support that have considerable impact on software design: reflectivity, attribute programming and events/delegates.

3.3.1 Introspection and Reflectivity[22] [41]

A program that can explicitly see, understand and modify its own structure is said to have introspective capabilities. Reflectivity is a property that a program may possess that permits its structure to be accessible to itself. The information that is accessible through introspection is called meta-information or meta-data. Meta-data permits the

creation of simple but powerful tools that help the design and development of software such as debuggers, class browsers, object inspectors and interpreters. There exist many languages such as Java and C# that are said to be reflective because they provide meta-information to programs written with them. Most reflective languages implement the reflection property by the means of a supporting run-time like the Java JVM or the .Net CLR, in this way separating the meta-information from the base program.

These concepts are illustrated in the reflection capabilities of the C# programming language where it is possible to query the CLI to know the structure of an object. To such a query, the CLI returns an object that is an instance of a metaclass named Type that fully describes the type. Table I gives a list of the basic classes that make metadata accessible to a program.

Table I : Metadata Classes

Class	*Description*
Type	Represents type declarations: class types, interface types, array types, value types, and enumeration types.
Assembly	Defines an Assembly, which is a reusable, versionable, and self-describing building block of a CLR application.
MethodInfo	Discovers the attributes of a method and provides access to method metadata.
ParameterInfo	Discovers the attributes of a parameter and provides access to parameter metadata.
FieldInfo	Discovers the attributes of a field and provides access to field metadata.
PropertyInfo	Discovers the attributes of a property and provides access to property metadata.
EventInfo	Discovers the attributes of an event and provides access to event metadata.
ConstructorInfo	Discovers the attributes of a class constructor and provides access to constructor metadata.
MemberInfo	Discovers the attributes of a member and provides access to member metadata.

Code example 1 exemplifies the use of some basic introspection classes to query a class about its members (fields, properties, constructors, methods, etc.)

```
1.  public class TypeIntrospection{
2.    public static void Main(){
3.      Type theType = Type.GetType("Assembly");
4.      MemberInfo[] mbrInfoArray = theType.GetMembers();
5.      foreach (MemberInfo member in mbrInfoArry)
6.        Console.WriteLine("{0} is a {1}", member,
7.          brInfo.MemberType);}}
```

Code example 1: Type Introspection

Excerpt of the ouput:

System.String s_localFilePrefix is a Field
Boolean IsDefined(System.Type) is a Method
Void .ctor() is a Constructor
System.String CodeBase is a Property

In line 3 we get a reference to the "Assembly" type. Line 4 retrieves all the members that are declared in the type. The rest of the code iterates through the members and prints them to the standard output.

Here is a code example that shows the true power of introspection and reflectivity. First, we dynamically discover and change the value of a private field, and then we dynamically discover and invoke an object's method.

```
1. public class MyClass{
2.   private string myString="Old value";
3.   public int MyStringLength(String inputString){
4.     return inputString.Length ;}}
5.
6. public class FieldInfo_SetValue{
7.   public static void Main(){
8.     MyClass myObject = new MyClass();
9.     Type myType = Type.GetType("MyClass");
10.    FieldInfo myFieldInfo = myType.GetField("myString",
11.      BindingFlags.NonPublic | BindingFlags.Instance);
12.    Console.WriteLine("\nField value of 'myString': {0}",
13.      myFieldInfo.GetValue( myObject ));
14.    myFieldInfo.SetValue( myObject, "New value",
15.      BindingFlags.Default, null , null );
16.    Console.WriteLine( "Field value of 'mystring' : {0}",
17.      myFieldInfo.GetValue( myObject ) );
18.    Object theObj = Activator.CreateInstance(myType);
19.    Type[] paramTypes = new Type[1];
20.    paramTypes[0] = Type.GetType("System.String");
```

```
21.   MethodInfo myMethod = myType.GetMethod
22.     ("MyStringLength",paramTypes);
23.   ParameterInfo[] pi = myMethod.GetParameters();
24.   Type returnType = myMethod.ReturnType;
25.   Console.WriteLine("The parameter type: {0}",
26.     pi[0].ParameterType);
27.   Object[] parameters = new Object[1];
28.   parameters[0] = "Hello";
29.   Object returnVal = myMethod.Invoke(theObj,parameters);
30.   int val = (int)returnVal;
31.   Console.WriteLine(val);}}
```

Code example 2: Value Modification with reflection

Excerpt of the ouput:

The parameter type: System.String
The return type: System.Int32
Field value of 'myString': Old value
Field value of 'myString': New value

Lines 10-11 show how to get a reference to the declaration of a private field by using the field's name. At line 13, we retrieve the value of the field for a particular object. Lines 14-15 demonstrate how to modify the value of the field for a particular object. Line 18 uses a static method of the *Activator* class to create an instance of a type. Lines 23-25 demonstrate how to discover the various aspects of a dynamically discovered class method. Lines 27-28 prepare the necessary parameters to make the dynamic call to the method and line 29 makes the call. This code fragment demonstrates the raw reflective powers that are missing in C++.

3.3.2 Attribute Programming[50] [41]

Both the C# and the CLI standards defined a method for adding declarative information (metadata) to runtime entities. Since the .Net Framework has at its core the CLI, it also has metadata support. The mechanism through which metadata may be added to a program is called attribute programming. Attributes can be added to all the elements of a program except the body of properties and methods. It is even possible to add declarative information to the assembly, which is a unit of deployment that is similar to an .exe or .dll file on the Windows platform.

As mentioned before, attributes in .Net may be used to add extra information about elements in a program but they also provide an elegant, consistent approach to adding declarative information to runtime entities that permit a new way of designing software. The mechanism to retrieve these attributes (metadata) at runtime has also been standardized, permitting software components developed by different teams or even companies to interact and discover each other through metadata. Metadata may even be used to control how the program interacts with different runtime entities. It is this capability that we exploit later in this thesis.

The following is an example of a possible attribute that could be used to tag a class with hardware type information. We give an example of a class tagged with some metadata and we recover the metadata using introspection.

```
1.  [HardwareType("CPU")]
2.  public class MyProcessor{
3.    public string technology= "FPGA";}
4.
5.  public class MetadataInspecter{
6.    public static void Main(){
7.      MyProcessor obj = new MyProcessor ();
8.      Type hardwaretype = typeof(HardwareTypeAttribute)
9.      Type type = obj.GetType();
10.     Object[] attributes =
11.       type.GetCustomAttributes(hardwaretype,false)
12.     foreach(Object attribute in attributes){
13.       HardwareTypeAttribute ht = attribute
14.          as HardwareTypeAttribute;
15.       Console.WriteLine("Hardware Type:{0}", ht.type);}}}
```

Code example 3: Attribute programming example

Excerpt of the ouput:

Hardware Type: CPU

The important lines are 10 and 11 which show how to retrieve custom metadata from a type object.

3.3.3 Delegates

Callbacks are an important concept in the implementation of event handling. Here is a good informal definition for the concept of a callback:

> A scheme used in event-driven programs where the program registers a subroutine (a "callback handler") to handle a certain event. The program does not call the handler directly but when the event occurs, the run-time system calls the handler, usually passing it arguments to describe the event. [29]

Most modern programming languages have constructs that permit the implementation of callbacks such as function pointers in C++ and interfaces in Java [2] . The .Net Framework and C# use delegates to address event handling. The concept of delegates improves upon function pointers by being object-oriented and type-safe and improves upon interfaces by allowing the invocation of a method without the need for inner class adapters. Also, delegates are inherently multicasting – a delegate contains a list of bounded methods that are invoked in sequence when the delegate is invoked. Another interesting difference between a delegate and a function pointer is that the delegate may contain an instance method in its invocation list, not only a static method as with function pointers, because the delegate keeps the information of the object that the method should be called on.

There are three steps in defining and using delegates: declaration, instantiation, and invocation.

Delegates are declared using delegate declaration syntax.

```
1. delegate void MyDelegate();
```

The example declares a delegate named `MyDelegate` that takes no arguments and returns no result.

Delegates are instantiated like all other object-oriented constructs.

```
1. class Test{
2.   static void F() {
3.     System.Console.WriteLine("Test.F");}
4.
```

```
5.   static void Main() {
6.     MyDelegate d = new MyDelegate(F);
7.     d();}} // delegate invocation
```

Code example 4: Simple delegate example

The example declares a variable of type *MyDelegate* and then instantiates it. The delegate is then invoked.

```
1. class Test{
2.   static void F() {
3.     System.Console.WriteLine("Test.F");}
4.
5.   static void G() {
6.     System.Console.WriteLine("Test.G");}
7.
8.   static void Main() {
9.     MyDelegate d = new MyDelegate(F); //static binding of F
10.    d += New MyDelegate(G) //dynamic binding of G
11.    d();}}// delegate invocation
```

Code example 5: Delegate example

In the above example, when the delegate is invoked, both the F and G methods are called in the sequence in which they were bound to the delegate.

C# has added a key to add event handling semantics to a class field that is a delegate type: *event.* A delegate qualified with the event keyword has no effect on the field from inside the class or class instance's scope. From outside the scope, however, the field may not be invoked, the field can only be used on the left-hand side of the += and –= operators. The += operator adds a handler for the event, and the -= operator removes a handler for the event.

```
1. public delegate void DataNotifyHandler(object sender,
2.   System.EventArgs e);
3.
4. public class DataProducer{
5.   public event DataNotifyHandler notify;}
6.
7. public class DataConsumer{
8.
9.   void DataReady(object sender, EventArgs e){
10.    Console.WriteLine("Data is ready!");}}
11.
```

```
12.  public class App{
13.
14.    static public void Main(){
15.      DataProducer prod = new DataProducer();
16.      DataConsumer com = new DataComsumer();
17.      prod.notify+= new DataNofityHandler(com.DataReady);
18.    }
```

Code example 6: Event keyword example

The above example shows a `DataConsumer` class that adds `DataReady` as an event handler for the **notify** event of a *DataProducer* class which is declared with the *event* keyword. This example shows how a simple and naive way of synchronizing a producer and consumer.

3.3.4 Delegates and Reflectivity

A powerfull combination is the use of reflection in collaboration with delegates. Compared to most language .Net/C# permits methods to be bound to a delegate at runtime. For example, in C++, the name of the method that is bound to a function pointer must be known at compile time, but in .Net/C# it is possible to create a delegate type object with a static method of the Delegate class. The method takes as parameters an object and the name of the method which should be bound to the created delegate.

With reflectivity, it is possible at runtime to discover the names of the various methods that an object supports, so it is possible to dynamically discover an object's method and bind it to a delegate.

The example below illustrates this flexibility:

```
1. public delegate void myMethodDelegate();
2.  public class MyClass{
3.    static public void Hello(){
4.      Console.WriteLine("Hello");}
5.    public void GoodMorning(){
6.      Console.WriteLine("Good Morning");}
7.    public void Bye(){
8.      Console.WriteLine("Bye");}}
9.
10. public class MainApp{
```

```
11.  static event  myMethodDelegate  myDelegate;
12.  static public void Main(){
13.   MyClass obj = new MyClass();
14.   Type objType = obj.GetType();
15.   Type myMethodDelegateType = typeof(myMethodDelegate);
16.   foreach(MethodInfo method in
17.    objType.GetMethods(BindingFlags.DeclaredOnly |
18.                       BindingFlags.Static |
19.                       BindingFlags.Instance |
20.                       BindingFlags.Public)){
21.    if(method.GetParameters().Length == 0 &
22.       method.ReturnType == typeof(void))
23.     if(method.IsStatic)
24.      myDelegate+=(myMethodDelegate)Delegate.CreateDelegate
25.                                  (myMethodDelegateType,method);
26.     else
27.      myDelegate+=(myMethodDelegate)Delegate.CreateDelegate
28.                          (myMethodDelegateType,obj,method.Name);
29.   }
30.   if(myDelegate!=null)
31.    myDelegate();}}
```

Code example 7: Delegates and reflection

This example iterates through all the static and instance methods that are declared public of an object (obj) of type *MyClass*. Each method is verified for two conditions: no formal arguments and a void return type. If the method fulfils the two conditions it is then bound to a delegate object (myDelegate). Static and instance methods of an object are bound dynamically to a delegate in different ways. Line 23-25 show how to bind a static method. Lines 26-28 show how to bind an instance method. Line 30-31 test the delegates in order to discover if it has been initialized. If it has a value other than null it is invoked.

The core of our environment makes use of an algorithm similar to code example 7, however, we incorporate the use of attributes to selectively filter the methods. Dynamic method discovery and delegate creation are useful because they enable a simple and elegant solution for implementing entry points in a simulation kernel for third-party tools. We also use them to dynamically create the processes of our simulation model (see Chap 5).

Chapter 4 ESys.Net

After struggling with the downfalls of SystemC, we looked for another alternative that would enable us to model systems in a simple effective manner and that would allow us to explore different types of CAD and EDA tools for partition, verification and synthesis. After looking at many environments and languages we stumbled cross the C# language and the .Net Framework. We immediately noticed that C# and .Net brought together several important features from various existing solutions i.e. Java, C++, ML, etc. and brought several new features that would probably enable the development of a new environment for system-level design based on the previous work of SystemC.

This section describes the fruit of our labor: **E**mbedded **Sys**tems with **.NET,** a new system-level design environment based on SystemC. ESys.Net is meant to be an evolution of SystemC by offering the same modeling capabilities but in a more elegant package. ESys.Net also innovates on SystemC by using a better underlying programming language which permits it to inherit operating system primitives, a rich software component library for rapid tool development and powerful runtime.

The next section will present briefly with the aid of an example the core elements of ESys.Net. These elements will then be explained in depth in subsequent sections.

4.1 A Simple Example

The best way to present a new tool is with an example, here is a simple example called "MyFirstSystem" that we will use to present our environment.

Here is a pictorial representation of MyFirstSystem.

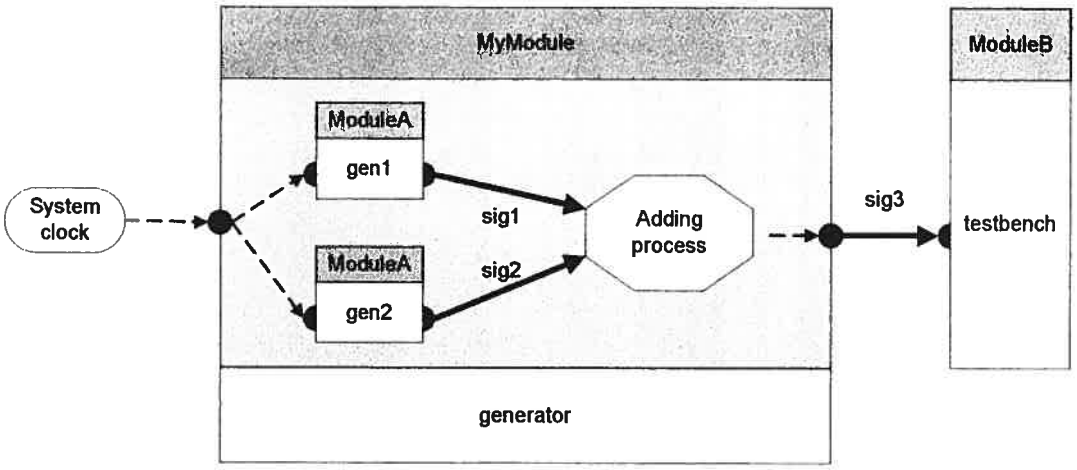

Figure 2 : My First System

"MyFirstSystem" is a model for a simple synchronous hardware component that is being tested with a testbench. The hardware component, named "generator", generates an integer on its output port on each positive edge of the main system clock. When a new value is generated by the hardware component, the testbench is notified of the new value by the "generator". The testbench then reads the new value from its input port and then prints it out.

Like most real hardware components, the "generator" is composed of sub-components. These sub-components are responsible for generating an integer value on each positive edge of the main system clock which is then added together by a computation process in the encapsulating component.

The following code represents the blueprint for the two sub-components (**gen1** and **gen2**):

```
1. public class ModuleA : BaseModule{
2.   public Clock clk;
3.   public outInt porta;
4.
5.   public ModuleA(): base(){}
6.
7.   [Process]
8.   [EventList("posedge","clk")]
9.   public void Gen() {
10.    while(true){
```

```
11.     for(int i=0;0<100;i++){
12.       porta.Value=i;
13.       Wait();}}}}
```

Code example 8: ModuleA blueprint

Code example 8 declares the specification for a component that has one input port, **clk** and one output port, **porta**. The input is used to drive the component with a clock signal. The output is used to transmit an integer value that is generated by the *Gen* method. Hardware elements are concurrent by nature, they all compute in parallel. To indicate that the ***Gen*** method represents a computation that is concurrent, which we call a process, we tagged the method with a ***Process*** attribute. All methods tagged with the ***Process*** attribute execute concurrently. The ***EventList*** attribute indicates that the ***Gen*** method is sensitive to the positive edge (**posedge**) of the **clk** input.

The role of the ***EventList*** tag is to indicate an association between a process and a triggering object which is implemented with the *Event* class that is defined in our environment. As its name implies, the *Event* class represents an event that may occur during the simulation of the model. When an event is triggered, the processes that are associated to an event are executed. In the above example, the **clk** field owns an event called **posedge** – that represents the event of a positive edge-, when the clock represented by **clk** field generates a positive edge, it triggers it **posedge** event.

The ***Wait*** method call in the ***Gen*** method indicates that the execution should stop at that point and then resume when an event on its triggers list occurs.

```
1. public class ModuleB : BaseModule {
2.
3.   public inInt porta;
4.   public Event syn_event;
5.
6.   public ModulB(): base(){}
7.
8.  [Process]
9.  public void Run() {
10.         while(true){
11.               Wait(syn_event);
12.               Console.WriteLine(porta.Value);}}}
```

Code example 9: ModuleB blueprint

This code fragment represents the blueprint of the testbench module. It only has an input port **porta**. Its *Run* method has been tagged with a ***Process*** attribute so it will run concurrently with the *Gen* methods of the **gen1** and **gen2** sub-components. What is distinct about this module is that the method indicated to become a process does not have an ***EventList*** attribute. This is not a problem, before the simulation starts, all methods that are processes that do not have an event list are executed once. The *Wait* method call with an event as an argument within the *Run* method indicates that the method will stop at this point and wait until the event is triggered.

```
1. public class MyModule : BaseModule {
2.  public outInt porta;
3.  public Clock clk;
4.  public Event syn_event = new Event();
5.
6.  ModuleA gen1 = new ModuleA();
7.  ModuleA gen2 = new ModuleA();
8.   IntSignal sig1 = new IntSignal();
9.   IntSignal sig2 = new IntSignal();
10.
11.  public MyModule(): base(){}
12.
13.  [PMethod]
14.  [EventList("sensitive","sig1","sig2")]
15.  public void Add() {
16.         porta.Value = sig1.Value + sig2.Value;
17.         syn_event.Notify(0);}
18.
19. public override void BindingPhase(){
20.         gen1.clk = clk;
21.         gen2.clk = clk;}}
22.
```

Code example 10: MyModule blueprint

This code fragment is the blueprint for the main module called "generator". It has an input for a clock signal and an output for an integer value. It also contains two sub-components of type *ModuleA* – the blueprint presented earlier-, two signals used to connect the sub-components together and an event instance that will be used to notify the testbench when the "generator" generates a new value. The component has a method that is tagged with the ***PMethod*** attribute. This indicates that the method

should be considered as a concurrent process like a method that is tagged with ***Process***. The difference between a method tagged with ***PMethod*** and ***Process*** is in their simulation implementation. A method tagged with a ***Process*** attribute keeps its state between *Wait* method calls. A method tagged with a ***PMethod*** tagged cannot call the *Wait* method, it is executed like a normal method call so the state of variables declared in the body of the method are not kept between execution.

The ***EventList*** of the *Add* method indicates that the method is sensitive to the two internal signals (**sig1** and **sig2**) that are used to communicate with the sub-components. Each signal owns an event called sensitive. A signal triggers its **sensitive** event when a new value that is different from its former value is written to the signal. The Notify method call in the *Add* method on the **syn_event** event causes the event to be triggered which causes the testbench module's *Run* method to be awaken and executed.

The last method in the "generator" component is called *BindingPhase*. It is invoked before the simulation starts and is used to propagate signals throughout a module hierarchy for binding purposes. Binding is necessary to connect a signal to a port such as the **clk** signal being bound to the **clk** ports of the two sub-components.

```
1. public sealed class MyFirstSystem:SystemModel{
2.
3.   TopModule top = new TopModule();
4.   ModuleB testbench = new ModuleB();
5.   Clock clk = new Clock("clock1",4);
6.   IntSignal sig3 = new IntSignal();
7.
8. public MyFirstSystem (ISystemManager manager): base(manager){
9.   top.clk = clk;
10.        top.porta = sig3;
11.        testbench.porta = sig3;
12.        testbench.syn_event = top.syn_event;}}
```

Code example 11: MyFirstSystem

The overall "MyFirstSystem" system is defined by creating a class that inherits from *SystemModel*. This class is used to encapsulate a system and is the entry point for the simulator to extract the model to be simulated. In this class we instantiate the main

module, the system clock, a signal and the testbench. The elements are then connected together, **sig3** is connected to the main module and the testbench, and the main module's **syn_event** event is connected to the testbench.

To execute the simulation, we need to create an instance of our model and an instance of a simulator; then we must bind the two together and start the simulator. Here is the code:

```
1.  public class MyApp{
2.
3.  public static void Main(){
4.    Simulator sim = new Simulator();
5.    MyFirstSystem sysModel = new MyFirstSystem (sim);
6.    sim.sysm = sysModel;
7.    sim.Run(50);
8.       }
9.  }
```

Code example 12: MyApp

Like Java, the execution entry point for a C# application is a public static method called *Main*. The 50 passed in the *Run* method of the **sim** instance is the number of time units that must be simulated before stopping. The other sections of this chapter will explain in more details each of the modeling concepts that were briefly presented in this example.

4.2 Modules and Module Hierarchies

As time passes, design problems are not becoming simpler, they are becoming larger and ever more complex. Luckily, one of the oldest approaches to complex problem solving is still alive and well: divide and conquer. The basic concept of subdivision in ESys.Net is a user defined module. Modules are like black boxes, we use them by connecting them together without knowing how they work. So a simple and informal definition of a module could be: an entity that encapsulates a service or a functionality that an end user may access through its user interface. The word interface, as used here, simply means the outer wrapper of the module that is available to the end user, for example the touch pad of a microwave is part of a user interface. By breaking down a complex problem into simpler ones, implementing

those simpler problems into modules and them encapsulating connected modules into higher level modules, we can create a hierarchy of modules that is simple to manage, to understand and more importantly solves a complex problem.

The power of abstraction that modules permit is very important. Since an end-user only relies on the interface of a module and not the internal workings of it, a systems designer may interchange, in a plug-and-play fashion, modules with the same interface to quickly modify his design. From the perspective of a module designer, the abstraction of implementation permits the designer to constantly upgrade the internal workings of his design without affecting the rest of the system that his module may be part of.

Complex systems are very difficult to model directly, so we usually partition them into simpler sub-systems that we can easily model and then recombine to form complete system. ESys.Net, like most existing HDLs, proposes the partitioning of complex systems into logical blocks possessing inputs, outputs and processing capacities.

There are two parts involved in the use of a user defined module:

1. The declaration of the module;
2. The instantiation of the module in the context of a system.

4.2.1 Module Declaration

Declaring a module is like creating a blueprint of a chip. A module declaration contains the blueprint of its interface (e.g. ports) and its inner workings (sub-components and processes).

```
1. public class MyModule: BaseModule{
2.   \\ Declared interface and inner components shouldd go here
3.   \\ but this is a simple portess ? component
4.
5.   \\ A simple constructor for anonymous components
6.   public MyModule(){}
7.   \\ Another simple construtor
8.   public MyModule(string name): base(name){}
9.   \\ Declared inter workings should comme here
```

```
10.  \\ but this component does not do anything.}
```

Code example 13: General module declaration

Since ESys.Net relies on an object-oriented framework, all user defined modules must inherit from the *BaseModule* class and implement the necessary constructors. If the constructor on line 3 is used when instantiating the user defined module, a default identification name will be generated for the module instance. If the constructor on line 4 is used, the programmer must supply the module identification name when instantiating the module. All module instance identification names in the context of a module hierarchy level must be unique, meaning that modules that are siblings of the same parent in the module hierarchy tree must have unique identification names.

4.2.2 Module Instancing

As a chip blueprint is only a design on paper, a user module declaration, which is a user defined class declaration, is the same. As with all statically type object-oriented programming languages, we must declare a variable of the same type as the need user-defined module and initialize the declared variable by calling a constructor.

```
1.  MyModule module_a = new MyModule("modulea");
2.  MyModule module_b = new MyModule();
```

Code example 14: Module instantiation

The code on line 1 initializes a reference variable called **module_a** by instantiating an object of type *MyModule* and assigns the instance identification name "modulea" to **module_a**. The code on line 2 initializes a reference variable called **module_b** by instantiating an object of type *MyModule* and assigns a generated instance identification name to **module_b**.

4.2.3 Module Hierarchies

As an IC may itself contain other sub-IC components, modules may themselves contain instances of other modules. Also, as a sub-IC component is not visible from outside of its containing IC; inner modules are usually not accessible from outside their container module either. We can see that by recursively creating modules by

composition, we obtain a hierarchy or tree of modules. Module hierarchies are obtained by instancing modules within module declarations.

```
1. public class ModuleA: BaseModule{
2.   \\ Declared interface and inner components shouldd go here
3.   \\ but this is simple portess component
4.
5.   \\ A simple construtor for anonymous components
6.   public MyModule(){}
7.   \\ Another simple constructor
8.   public MyModule(string name): base(name){}
9.   \\ Declared inter workings should comme here
10.  \\ but this component does not do anything.}
11.
12.public class ModuleB: BaseModule{
13.  \\ These are sub-components instantiation
14.  ModuleA mod1 = new ModuleA("mod1")
15.  ModuleA mod2 = new ModuleA("mod2")
16.  \\ A simple constructor for anonymous components
17.  public ModuleB(): base(){}
18.  \\ Another simple constructor
19.  public ModuleB(string name): base{}
20.  \\ declared inter workings should comme here
21.  \\But this component does not do anything.}
```

Code example 15: Module hierarchy

Sub-modules should be declared private because they should be semantically hidden within the scope of their parent module – not accessible from outside their containing module. The keyword private was omitted in the above code because by default, C#'s variables are private and not public like Java.

4.2.4 Module Interfaces

A module's interface is the only thing exposed to the outside world; the basic element used to make a module's interface is a port. A port represents an entry or exit point for data moving in or out of the module.

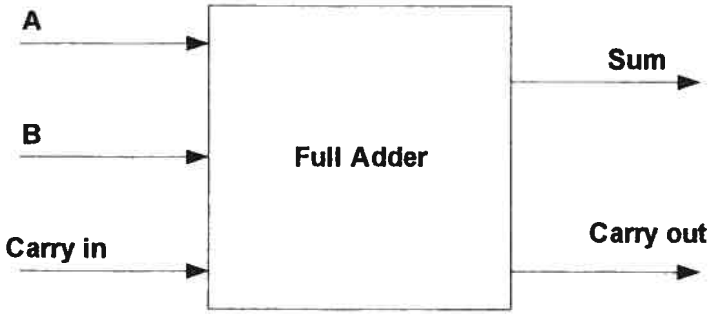

Figure 3 : Full Adder Module Interface Example

Figure 3 shows a FullAdder module with a number of ports. The ports on the left are input ports or in/out ports while the ports on the right are output ports. Each port has an identification name.

The above module and its interface could be declared as follows:

```
1. public class Fifo: BaseModule{
2. public inBool A;
3. public inBool B;
4. public inBool CarryIn;
5.
6. public outBool Sum;
7. public outBool CarryOut;
8.
9. public FullAdder (): base(){}
10. public FullAdder (string name): base(name){}
11. }
```

Code example 16: FIFO declaration

4.2.5 Modules Inner-Workings

The semantics of modules and interfaces as mentioned above permits the partitioning of a system into logical blocks but what is important above all is the processing capabilities of the block.

The logical unit of processing in ESys.Net is a process. Since hardware by nature is inherently very parallel and that one of our main objectives is to simulate hardware components, a process is simulated in parallel with all the other processes that makeup the simulated model. A basic process is simply declared by creating a standard private class method that takes no parameters, returns nothing and is tagged

with a ***Process*** and ***EventList*** attribute. Since a process is basically a class method, it has access to class variables, instance variables and other class methods. Also, since the ports of a module are instance variables, processes have access to them and it is by this mechanism that a module can read from the input ports of a module and write on the output of the module.

Here is an example of a simple one bit adder:

Figure 4 : One Bit Adder Example

```
1. public class FullAdder: BaseModule{
2.  public inBool a;
3.  public inBool b;
4.  public inBool carryIn;
5.  public outBool sum;
6.  public outBool carryOut;
7.  public FullAdder (): base(){}
8.  public FullAdder (string name): base{}
9.
10.
11.   [Process]
12.   [EventList("sensitive", "a", "b", "carryIn")]
13.   private void Add(){
14.         bool tmp = (a.value ^ b.value);
15.         sum = tmp ^ carryIn.value;
16.         carryOut = (a.value & b.value) | (tmp & carryIn.value);
17.   }
18. }
```

Code example 17: Adder implementation

Lines 2 to 6 declare the Adder's interfaces. Line 12 to 16 declares the Adder's add process. The tag on line 10 is associated with the method that follows and indicates that it is a process of type process method (Section 4.2). The tag on line 11 indicates

the process is sensitive to ports **a**, **b**, and **carryIn** – meaning that the process should be called when there is a value written on ports **a, b, c**. Since ESys.Net is based on an event-driven simulation, it is necessary to indicate when a process should be called; if the inputs of a process don't change, the outputs should not change either so there is no use in executing the process for nothing.

4.3 Processes

Systems are all about processing data. Even with elegant solutions for system decomposition or abstracting, system modeling solutions are useless without solid data processing constructs. The logical unit of data processing in ESys.Net is a process. From a semantics point of view, processes are non hierarchical entities that transform data in parallel. Since a system in ESys.Net is partitioned into black box components that take data as inputs and produce data as outputs, a process is always contained within a module.

A process is created by tagging a module's private method with a custom attribute predefined in our environment. When a model is discovered and analyzed before simulation, the ESys.Net kernel detects via reflexivity all private methods that are tagged with the necessary attributes and registers those methods to be managed by the kernel.

As previously explained, since ESys.Net is based on an event-driven simulation kernel, all the actions to be performed are linked to an event. When the event is triggered the associated actions are performed in parallel. Since processes represent the actions in our environment, they are all associated to events in a one-to-many fashion. We will see that there are two kinds of process-event associations: static and dynamic.

4.3.1 Process Declaration and Registration

In order to be eligible to become a process, a class method must have a private scope, have no formal arguments and return nothing. There are two kinds of processes in ESys.Net, which will be explained later, process methods and parallel methods

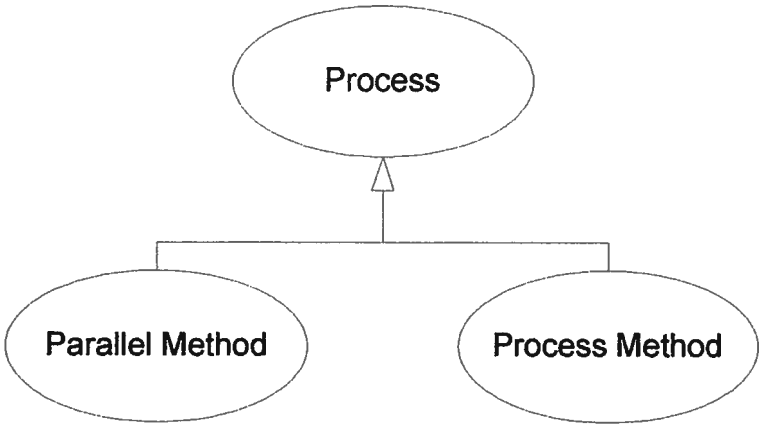

Figure 5 : Process Sub-Types

In order to indicate that a declared method should be considered by the kernel to be a process method or parallel method, the method is tagged with the ***Process*** or the ***PMethod*** attribute respectively.

```
1. public class ModuleA : BaseModule {
2.
3. public ModuleA(): base(){}
4. public ModuleA(String name): base(name){}
5.
6. [Process]
7. private void aProcessMethod() {}
8.
9. [PMethod]
10. private void aParallelMethod() {}
11. }
```

Code example 18: Process method and Parallel method declaration

4.3.2 Static and Dynamic Process-Event Association

A process can be bound to an event in one of two ways: statically and dynamically. Semantically, a static link between a process and an event means that the process-event association holds globally for the entire simulation; whereas a dynamic association holds only for a certain period of time. Also, a process' static association list is declared with the process' declaration, whereas a process' dynamic association list changes throughout the execution of the simulation. In ESys.Net a process' dynamic association list has precedence over its static association list - if a process has at a certain point in time a non empty dynamic association list, the process will

not be triggered by events in its static association list until the process' dynamic list is empty.

4.3.3 Process Static Sensitivity

A process' static event association list is created using the ***EventList*** tag on a method already tagged with a ***Process*** or ***PMethod*** tag. The ***EventList*** tag takes as its first argument the name of an event that we want to associate with the process, and has as remaining arguments an unlimited number of entities (signals, channels, etc...) which are owners of an event that has the specified name in the first argument.

```
1. public class ModuleA : BaseModule {
2. public IntSignal sig1 = new IntSignal();
3. public IntSignal sig2 = new IntSignal();
4.
5. public ModuleA(): base(){}
6. public ModuleA(String name): base(name){}
7.
8.
9. [Process]
10.  [EventList("sensitive","sig1","sig2")]
11.  private void aProcess() {
12.        …
13.  }
```

Code example 19: Static Sensitivity

The above example states that the module has a process method that must be statically associated with two events which have the name "sensitive", one owned by **sig1** and the other owned by **sig2**.

If the process method must be associated with events of different names, multiple ***EventList*** tags must be used for each event name.

```
1. public class ModuleA : BaseModule {
2. public IntSignal sig1 = new IntSignal("sig1")
3. private Event myEvent = new Event();
4. public ModuleA(): base(){}
5. public ModuleA(String name): base(name){}
6.
7. [Process]
8. [EventList("sensitive","sig1")]
```

```
9.  [EventList("myEvent","this")]
10.  private void aProcess() {
11.        …
12.  }
```

Code example 20: Static Sensitivity with multiple events names

The above example states that the module has a process method that must be statically associated with two events, one owned by **sig1** with the name "sensitive" and the other named "myEvent" owned by the current module (the owner name ***this*** is a keyword meaning the current module).

4.3.4 Parallel Method Process

A parallel method (pmethod) process is executed by the ESys.Net kernel as a synchronous method invocation, so upon completion it returns control to the ESys.Net kernel. Because of its implementation, a pmethod does not maintain the state of its local variables and it is impossible to explicitly suspend the pmethod's execution – it may not have calls to the *Wait* method or have an infinite loop. If the state of the local variables must be kept between pmethod invocations, the user must explicitly manage them using class variables.

A pmethod's dynamic sensitivity list is created using the *NextTrigger* method with one or more event objects as arguments. The *NextTrigger* method may be called in the body of the pmethod process code, or it may be called in a method called by the pmethod that is either owned by the current module or communication channel.

Parallel Method Dynamic Sensitivity

When triggered, the entire body of the pmethod is executed. Execution of a *NextTrigger* statement sets the sensitivity for the next triggering event of pmethod. The execution of *NextTrigger* does *not* cause the pmethod to end prematurely. The *NextTrigger* method specifies the event, event list or time delay that is the next triggering condition for the pmethod. If multiple *NextTrigger* statements are executed, the last one executed before the pmethod completes determines the next trigger condition (i.e. last one wins).

```
1. public class ModuleA : BaseModule {
2. public IntSignal sig1 = new IntSignal("sig1")
3. private Event myEvent = new Event();
4.
5. public ModuleA(): base(){}
6. public ModuleA(String name): base(name){}
7.
8.
9. [PMethod]
10. [EventList("sensitive","sig1")]
11. private void aProcess() {
12.       NextTrigger(myEvent);
13. }
```

Code example 21: PMethod Dynamic Sensitivity

4.3.5 Process Method

A process method is implemented with a .Net Framework thread. The process method runs until a *Wait* method call is executed whereupon the process is suspended. Upon suspension the state of the process is implicitly saved. The process method is resumed based upon its sensitivity list. Its state is then restored and execution of the process method resumes from the point of suspension (statement following *Wait*).

If the body or parts of the body of the process method are required to be executed more than once then it must be implemented with a loop, typically an infinite loop. This ensures that the process can be repeatedly reactivated.

If a process method does not have an infinite loop and does not call *Wait* in any way then the process will execute entirely and exit within the same delta-cycle. The *Wait* method can be called in the body of the process method code, or can be called in a method called by the process method that is either of a member function of the module or a method of a channel.

If a process method does have an infinite loop but does not call *Wait* in any way then the process will continuously execute during the same delta-cycle. No other process will execute (ESys.Net currently executes one process at a time to mimic SystemC; the next version will execute multiple processes).

Process Method Dynamic Sensitivity

Execution of the *Wait* method with arguments specifies the condition or conditions for resuming the process method. This list of arguments is considered the process method's dynamic sensitivity list.

```
1. public class ModuleA : BaseModule {
2. public IntSignal sig1 = new IntSignal("sig1")
3. private Event myEvent = new Event();
4.
5. public ModuleA(): base(){}
6. public ModuleA(String name): base(name){}
7.
8.
9. [Process]
10.  [EventList("sensitive","sig1")]
11.  private void aProcess() {
12.        while(true){
13.              Wait(myEvent);
14.        }
15.  }
```

Code example 22: Process Method dynamic sensitivity

Empty Static Sensitivity List

If a pmethod or process method has no static sensitivity list specified then it will be automatically executed once before the simulation starts.

Triggering on a single event

If the *NextTrigger or Wait* method is called with a single event argument then the pmethod will be triggered when that event is triggered. Syntax for triggering on a single event:

```
1. private Event myEvent = new Event();
2.
3. NextTrigger(myEvent);
```

```
4. Wait(myEvent);
```

Code example 23: Triggering on a single event

Triggering after a specific amount of time

If the *NextTrigger or Wait* method is called with a time value argument then the process will be triggered after a delay of the specified time. Syntax for triggering after a specific amount of time:

```
1. NextTrigger(200);
2. Wait(200);
```

Code example 24: Triggering after a specific amount of time

If the time value argument is zero then the process will be triggered after one delta-cycle. Syntax for triggering after one delta-cycle delay:

```
1. NextTrigger(0);
2. Wait(0);
```

Code example 25: Triggering with zero time

Triggering on one event in a list of events

If the *NextTrigger or Wait* method is called with an OR-list of events then the process will be triggered when one event in the list of events has been triggered. Syntax for triggering on one event in a list of events:

```
1. NextTrigger(e1 | e2 | e3);
2. Wait(e1 | e2 | e3);
```

Code example 26: Triggering on one event in a list of events

Triggering on all events in a list of events

If the *NextTrigger or Wait* method is called with an AND-list of events, then the process will be triggered when all events in the list of events have been triggered. The events do not have to be triggered in the same delta-cycle or at the same time. Syntax for triggering on all events in a list of events:

```
1. NextTrigger(e1 & e2 & e3);
```

```
2. Wait(e1 & e2 & e3);
```

Code example 27: Triggering on all events in a list of events

Triggering on an event in a list of events with timeout

If the *NextTrigger or Wait* method is called with a combination of a specific amount of time and an OR-list of events, then the process will be triggered when one event in the list of events has been triggered or after the specified amount of time whichever occurs first. Syntax for triggering on one event in a list of events with timeout:

```
1. NextTrigger(200, e1 & e2 & e3);
2. Wait(200, e1 & e2 & e3);
```

Code example 28: Triggering on an event in a list of events with timeout

Triggering on all Events in a list of events with timeout

If the *NextTrigger or Wait* method is called with a combination of a specific amount of time and an AND-list of events then the process will be triggered either when all events in the list of events have been triggered or after the specified amount of time which ever occurs first. Syntax for triggering on all events in a list of events with timeout:

```
1. NextTrigger(200, e1 | e2 | e3);
2. Wait(200, e1 | e2 | e3);
```

Code example 29: Triggering on all Events in a list of events with timeout

4.4 Signals

With the concepts of modules and processes, it is possible to break up a complex problem into logical sub-units of functionality and describe the parallel processing entities that they contain. We are, however, not able to describe the interconnections that must exist to assemble the numerous modules together. Signals are the basic entities that permit interconnections between modules. They play the same role as wires and PCB traces, but they also play a more complex and deeper role in our simulation.

4.4.1 Signals and Simulation

In order to create a usable simulation model of a system, there are two missing concepts required to glue everything together. Firstly, we need a way to transport information between modules and secondly, all data moving from one module to another must happen or seem to happen in parallel. To fulfill the first requirement, we need a container that stores information moving to and from module ports. The second requirement is a little more complex to satisfy because the amount of parallelism available on a typical computer is much lower than needed to simulate hundreds of data items moving in parallel. As a result, we must use a software management system to simulate the parallel movement of data. In order to achieve our second requirement, we use a concept called the delta cycle. All information moving out of a module at a current delta cycle will only be available at the next delta cycle.

Signals fulfill our two missing requirements. They are containers for information travelling between modules and they are the buffers that help implement the delta cycle concept. When a module writes a value to a signal, the signal stores the value but does not make it accessible until the next delta cycle. This implies that even if a module writes to a signal before another module can read the current value - this situation occurs because we cannot effectively execute the read and write in parallel so they are serialized in a delta cycle-, the value read by the module is the one that was current at the beginning of the current delta cycle.

4.4.2 Instancing

In ESys.Net, there is a class that represents a signal for every basic type supported by the CTS. To transport other types of data, there is a signal that manipulates objects and since the CTS is based on a unified type system, we can use it to transport any data of a user defined type (when reading the value, however, we must cast). Like all the elements in ESys.Net, signals have two constructors: one that lets the user specify the instances identification name and another that generates the names automatically.

```
1. IntSignal sig1 = new IntSignal("sig1");
```

```
2. IntSignal sig2 = new IntSignal();
```

Code example 30: Signal Instancing

The code on line 1 declares and initializes a reference variable called **sig1** by instantiating an object of type *IntSignal* -which is a signal that transports an Integer datum- and assigns the instance identification name "sig1" to **sig1**. The code on line 2 declares and initializes a reference variable called **sig2** by instantiating an object of type *IntSignal* and then assigns a generated instance identification name to **sig2**.

4.4.3 Inner and Outer Signals

A signal can either be visible on both sides of a module's boundary -the signal is used from within the module and is used by the module's outer environment- or it may only be visible from within the boundaries of its owning module. It is important to note that a signal may be an "inner" signal in one reference but an "outer" signal in a lower hierarchal reference (e.g. a signal that is only used from within a module is considered as "inner" for the reference of that module but if it is used by one of the module's sub-modules then in the reference of a sub-module the same signal is considered to be "outer") and also that a signal will only be considered "inner" in one reference. In Figure 6 we can see that the **InnerA** signal is considered inner in the presented reference of the **ModuleA** but would be considered "outer" if we took the reference of the **SM1** sub-module.

We can further illustrate the concept of an outer and inner signal with the example of an IC. A wire that is visible only from within an IC is considered "inner" but if a signal is also connected to the outer pin of the IC, it is visible from the outside also so it is considered "outer".

4.4.4 A Signal's Logical Scope

Like a variable, a signal has a logical scope. The scope determines how it is declared and instantiated. However, unlike a variable, a signal has many scopes because it has a scope for every module that uses it. Using the concepts of "inner/outer" signals as references, we can alternatively say that a signal has a scope for every reference in which it exists and that scope can be "inner" or "outer". The concept of scope is very

important because we must determine in which modules a signal is used, for a variable that represents the signal must be declared in all modules that use it. What is truly important is that we can only instantiate one signal that we assigned to the declared variables representing it.

In order to illustrate the concept of a signal's scope we shall use an example. Figure 6 illustrates a simple module – **ModuleA**- composed of 2 sub-modules – **SM1** and **SM2** - and a process – **P1**. The top module has 3 input signals – **InA, InB** and **InC** – 3 outputs signals – **OutA**, **OutB** and **OutC**- and 3 inner signals – **InnerA**, **InnerB** and **InnerC.** The **InA** signal feeds the **SM1** sub-module to produce the **InnerA** signal which feeds the **SM2** sub-module to produces the **OutA** signal. The SM2 sub-module is also fed by the **InnerB** signal produced by the **P1** process with the **InB** signal. The **P1** process also produces the **OutA** signal. The **InC** signal also feeds the **InnerC** that then feeds the **OutC**. If we take signal **InnerA** for our study of scope, we can see that it is used by modules **ModuleA** (which encapsulates it), **SM1** (which uses it) and **SM2** (which uses it). **InnerA** has a scope in all three modules, so each module has a local variable declared as the same type as **InnerA**

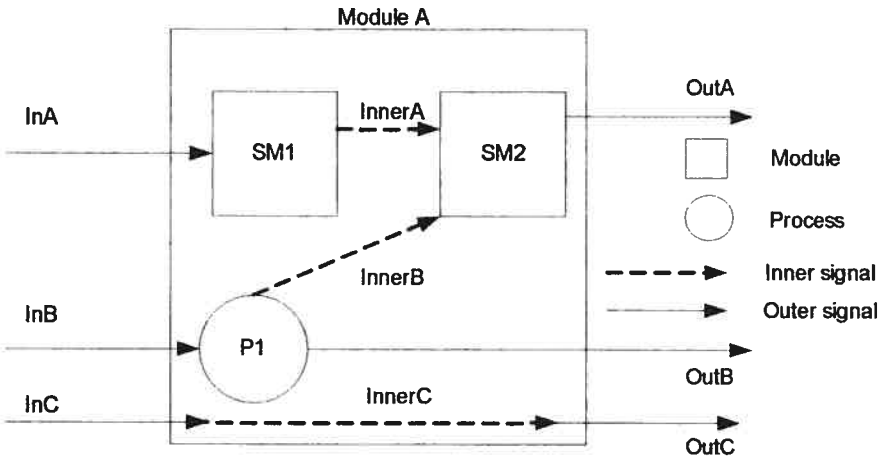

Figure 6 : Inner/Outer Signals

With the concepts of a signal's scope and "inner/outer" signals, we can state that a signal must be declared private and instantiated in its scope where it is considered "inner". It must be declared public in all scopes where it is considered "outer" and

initialized (bound) with the instance created in the "inner" scope. Another reason an "outer" signal must be declared public (besides the fact that we must have access to it to initialize it) is that the signal is a gateway between a module's "inner" and its "outer" world, thus it must be visible on both sides to fulfill this role.

Here is one possible code for the illustrated example.

```
1. public class ModuleA : BaseModule {
2.   …
3.   SM1 sm1 = new SM1();
4.   SM2 sm2 = new SM2();
5.  IntSignal InnerA = new InnerA();
6.   …
7.   sm1.InnerA = InnerA; //binding
8.   sm2.InnerA = InnerA; //binding
9.  …
10.  }
11.
12.  public class SM1 : BaseModule {
13.  …
14.  public IntSignal InnerA;
15.  …
16.  }
17.  public class SM2 : BaseModule {
18.  …
19.  public IntSignal InnerA;
20.  …
21.  }
```

Code example 31: Inner/Outer signals

4.4.5 Special Binding Cases

There exist two special binding cases which need particular attention: binding an "outer" signal to an "inner" signal and binding an "outer" signal to a sub-module's "outer" signal.

The first case presents the following problem: when we bind an "inner" signal to an "outer" we are basically extending the "inner" signal with the "outer" signal (or vice-versa). If we follow the rules mentioned above we should declare a variable for the "outer" signal, then we should declare and instantiate a variable for the "inner" signal. The problem is that we cannot glue the two signals together in order to propagate the

value from one to the other. Also, when the constructor of the module containing the "outer" signal and the "inner" signal is called, the "outer" signal is not initialized at that moment (it is only bound later); hence we don't even have two signals to glue together and we cannot bind the "inner" signal to the "outer" because an "outer" signal should always receive its value from the "outer" environment.

The second case is almost the same as the first; it occurs when we want to bind an "outer" signal to the "outer" signal of a sub-module. The problem is that when the constructor of the parent is called the sub-module is created but the parent's "outer" signal does not have a value at that current moment, so it can not be propagated to the sub-module.

In order to solve the two state problems, there is a virtual method that all user modules inherit which is called after all the module hierarchy is created, the *BindingPhase* method. If a user module does not have any of these special cases, overriding the method is not necessary and all the instancing of the signals (and sub-modules) and binding to the sub-modules may be done in the module's constructor. If the special cases are present, the user must override the method and put the signal binding code there. For the first case, no signal is instantiated for the "inner" signal and it receives its value from the "outer" signal. It is, however, impossible to glue the "outer" signal belonging to the same module with an "inner" signal; as a result, in Figure 6, the route from **InC** to **InnerC** to **OutC** is impossible to create without a process separating the inner signal from one of the outer signals.

In the second case, the sub-module's "outer" signal is bound with the value of the module's "outer" signal. This solution is possible because at the top most level, there should only be "inner" signals and they should have been bound to the modules at that level; so if we call the binding method after the creation of the module hierarchy it is possible to propagate the signal from the top to bottom.

Here is an example of the **InA** signal and the **InnerC** (with the premise of an added process between the **InnerC** signal and the **OutC** signal) signals from Figure 6.

```
1. public class ModuleA : BaseModule {
```

```
2. public IntSignal InA;
3.   public IntSignal InB;
4.   public IntSignal InC;
5. public IntSignal OutA;
6.   public IntSignal OutB;
7.   public IntSignal OutC;
8. IntSignal InnerA;
9.   IntSignal InnerB;
10.  IntSignal InnerC;
11.  SM1 sm1;
12.  SM2 sm2;
13.
14.  public ModuleA(): base(){}
15.  public ModuleA(string name): base{
16.  sm1 = new SM1();
17.  sm2 = new SM2();
18.  InnerA = new IntSignal();
19.  InnerB = new IntSignal();
20.  sm1.a_out_outer_signal = InnerA;
21.  sm2.a_in_outer_signal = InnerA;
22.  }
23.  …
24.  //the binding method
25.  public overrides void bindingPhase(){
26.  sm1.a_in_outer_signal = InA;
27.  sm2.a_out_outer_signal = OutA;
28.  InnerC = InC;
29.  }
30.  …
```

Code example 32: Special signal binding cases

We must point out that it is impossible to connect two outer signals belonging to the same module without using at least an intermediate process to copy the value of one signal to the other.

4.5 Ports and Interfaces

In most HDLs, ports are entities that makeup the interface of a module. Ports are like the pins of an IC, permitting the flow of "information" in and out of the module. It is through the concept of ports that modules can interact with their environment. In ESys.Net, the concept of ports does not explicitly exist. Ports are replaced by another concept of higher abstraction, a software interface. In this context, the word interface

has the same meaning as the concept of interfaces in object-oriented programming languages such as Java and C#. A software interface is composed of a set of method declarations but provides no implementation for those methods. Unlike Java, C# permits the declaration of properties (fields) to be part of an interface [2] . Software interfaces are then implemented by user defined types (classes), forcing the user defined type to implement a body for each method declaration in each software interface it uses [2] .

Software interfaces permit contractual programming and information hiding. In this way, a consumer of a reference variable declared as being of a certain interface type is guaranteed that the reference legally supports the set of methods declared in the interface definition and that no other methods are available.

4.5.1 The Elimination of Ports

We have eliminated the explicit concept of ports because in most HDLs a port is just an entity that adds an abstraction layer to a communication entity (like a signal), and offers a simple "read/write" API to access it. The proof of this is that in most HDLs, it is necessary to bind a signal to a port and a "read/write" to the port causes the "read/write" from the signal. The port is just delegating the work to the signal. Note that the functionality provided by a port is the same as a software interface; it is for this reason that we have eliminated ports.

In ESys.Net ports are used to control the way we access "outer" signals. Returning to the concepts of "inner/outer signals"; we can say that a signal which is "inner" in a certain scope can be logically accessed for reading *and* for writing. However a signal which is "outer" in a certain scope does not have both of its ends in the same scope so it can logically only be accessed for reading *or* for writing. The problem here is that signals implement both reading and writing functionalities, so it is very difficult to enforce which can be done and when. Since software interfaces permit contractual programming and information hiding, we can hide an "outer" signal declaration behind an interface that only supports the communication direction which is logical for that signal's current declaration scope. In addition, like the signal that the

software interface is hiding, it must be declared visible (public in C#) to the exterior world because it becomes the gateway to the outside world.

4.5.2 Predefined Interfaces

The ESys.Net framework provides a collection of predefined software interfaces. For each primitive value type supported by the CTS, there is an **in, out** and **inout** interface. The replication of the basic directional interfaces for every type is necessary because .Net and C# do not support templates or generics in their present state. The next version of .Net should have generics [37] so the interface library will be reduced to a collection of three generic interfaces: **in, out** and **inout**. Here is the model for the three basic directional interfaces but typed for a **Boolean** value.

```
1. public interface inBool {
2. bool Value{get;}bool IsChanged{get;}}
3. public interface outBool{
4. bool Value{set;}bool NextValue{get;}}
5. public interface inoutBool : inBool, outBool{}
```

Code example 33: Boolean software interfaces

Line 1 declares an interface called *inBool* that has two properties that are Booleans: one called "Value" that is read-only and one called "IsChanged" that is also read-only. Line 2 declares an interface called *outBool* that has two properties that are Booleans: one called "Value" that is set-only and one called "NextValue" that is read-only. Line 3 declares an interface called *inoutBool* that is the union of the *inBool* and *outBool* interfaces.

All the predefined software interfaces for ports in ESys.Net are based of the presented three interfaces. We have added the "IsChanged" and "NextValue" properties for verification and transactional support reasons. The "isChanged" property permits querying a signal hiding behind an interface in order to discover if it was modified during the preceding delta cycle, the "NextValue" property enables accessing the value that will be available on the signal during the next delta cycle.

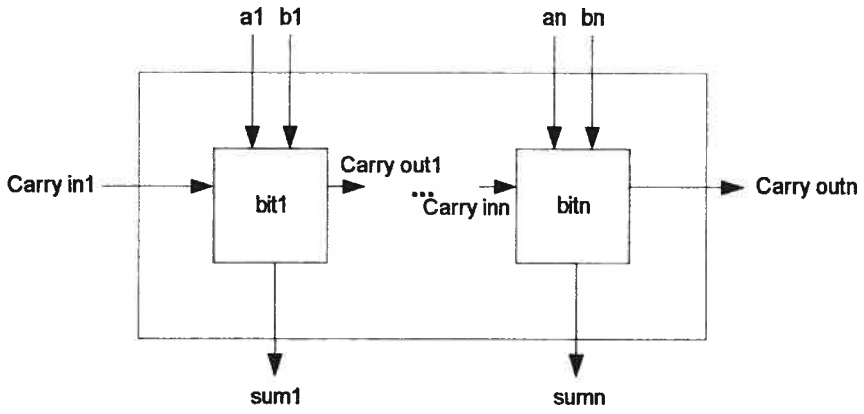

Figure 7 : A N bit Adder

Table II below shows two partial implementations of the *n* bit adder presented in the Figure 7 above. One implementation uses public signals for its interface; the other uses software interfaces to hide the signals in order to control how they are accessed.

Table II : Interfaces

Without the use of interfaces	With the use of interfaces
1. public class NBitAdder: BaseModule{	1. public class NBitAdder: BaseModule{
2. //interface declaration	2. //interface declaration
3. public BoolSignal[] a;	3. public inBool[] a;
4. public BoolSignal[] b;	4. public inBool[] b;
5. public BoolSignal[] sum;	5. public outBool[] sum;
6. public BoolSignal carryIn;	6. public inBool carryIn;
7. public BoolSignal carryOut;	7. public outBool carryOut;
8.	8.
9. //sub-modules decl. and int.	9. //sub-modules decl. and int.
10. FullAdder[] adder;	10. FullAdder[] adder;
11.	11.
12. //inner signals decl.and int.	12. //inner signals decl.and int.
13. BoolSignal[] innerCarry;	13. BoolSignal[] innerCarry;
14. int size;	14. int size;
15.	15.
16. public NBitAdder(int size): base(){	16. public NBitAdder(int size): base(){
17. a = new BoolSignal[size];	17. a = new BoolSignal[size];
18. b = new BoolSignal[size];	18. b = new BoolSignal[size];
19. sum = new BoolSignal[size];	19. sum = new BoolSignal[size];
20. innerCarry = new BoolSignal[size-1];	20. innerCarry = new BoolSignal[size-1];
21. adder = new FullAdder[size];	21. adder = new FullAdder[size];
22. for(int i = 0;i<size;i++)	22. for(int i = 0;i<size;i++)
23. adder[i] = new FullAdder();	23. adder[i] = new FullAdder();
24. this.size = size;}	24. this.size = size;}
25. public NBitAdder(int size,string na):	25. public NBitAdder(int size,string na):
26. base(name){this(size);}	26. base(name){this(size);}
27.	27.
28. public override void BindingPhase(){	28. public override void BindingPhase(){
29. for(i=0;i<size-1;i++){	29. for(i=0;i<size-1;i++){
30. adder[i].a = a[i];	30. adder[i].a = a[i];

Without the use of interfaces	With the use of interfaces
31. adder[i].b = b[i]; 32. adder[i].sum = sum[i]; 33. adder[i].carryIn = innerCarry[i]; 34. adder[i+1].carryOut= innerCarry[i]; 35. } 36. adder[i].carryIn = carryIn; 37. adder[size-1].carryOut = carryOut;}}	31. adder[i].b = b[i]; 32. adder[i].sum = sum[i]; 33. adder[i].carryIn = innerCarry[i]; 34. adder[i+1].carryOut= innerCarry[i]; 35. } 36. adder[i].carryIn = carryIn; 37. adder[size-1].carryOut = carryOut;}}

4.6 Events

Events play a crucial role in the ESys.Net environment because it is based, like SystemC, on an event-driven simulator. Events encapsulate the concept of an instance of time, the instance dwelling in the timeline of a simulation, and a group of actions (processes) to be performed at that time instance. Events also have a triggering cause that associates them to a specific instance of time. The triggering causes can be just about anything: the occurrence of a specific time in the simulation, the changing of a signal's value, the occurrence of another event etc.

A more practical view of an event in ESys.Net would be: events determine when a process execution should be triggered or resumed. An event is an object used to represent a condition that may occur during the course of simulation and to control the triggering of processes. When an event is notified (triggered), it causes the simulation kernel to execute the processes that are bound to the triggered event.

All events are instances of the Event class which is part of the ESys.Net framework.

```
1. Event myEvent = new Event(); // event declaration and
2. instantiation
```

Code example 34: Event instantiation

4.6.1 Event Occurrence

It is important to distinguish an event from the actual occurrence of that event. An event may have multiple occurrences, each occurrence being unique though reported through the same event. We can say that an event is like a conceptual relation between a point in time and a group of actions, and that an instance (or occurrence) of that relation links a specific moment of time to a specific group of actions.

An event is always owned. It may be owned by a module, channel or signal; it can also be global to a model, making it owned by all modules and channels in the model.

The owner of an event is responsible for creating an occurrence of the event (by notification) when the triggering cause of the event occurs (change of state of a channel, occurrence of a specific time in the simulation etc.). The event object, in turn, is responsible for keeping a list of processes that are linked to it. Thus, when notified, the event object will inform the simulation kernel of which processes to trigger.

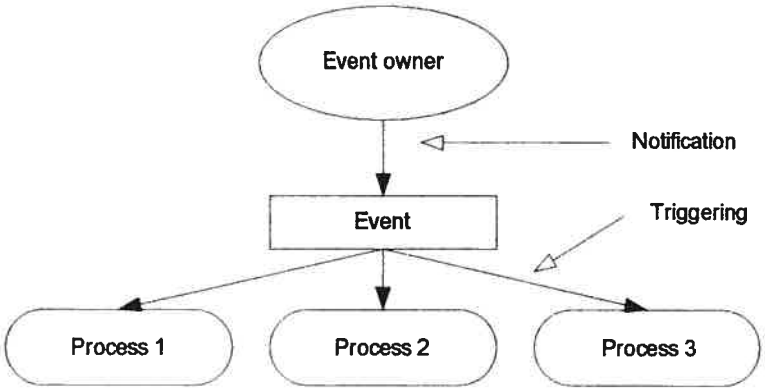

Figure 8 : Event Occurrence

4.6.2 Event Notification [63]

Events can be notified in three ways using its Notify method– immediate, delta-cycle delayed and timed:

- Immediate notification means that the event is triggered in the current evaluation phase of the current delta-cycle. The notify method with no arguments (Notify()) indicates immediate notification.
- Delta-cycle delayed notification means that the event will be triggered during the evaluation phase of the next delta-cycle. The notify method with an argument of zero indicates a delta-cycle delayed notification - the event is scheduled for the next delta-cycle.
- Timed notification means that the event will be triggered at the specified time in the future. The notify method with a non-zero argument (Notify(x))

indicates a timed notification of x simulation time units. The time of notification is relative to the execution time of the notify method as opposed to an absolute time.

```
1. 1.      Event myEvent = new Event(); // event declaration and
2. instantiation
3. 2.      myEvent.Notify(); // immediate notification
4. 3.      myEvent.Notify(0); // delta-delay notification
5. 4.      myEvent.Notify(10); // declaration of a 10 time unit
6. notification
```

Code example 35: Event notifications

Lines 2 to 4 are occurrences or instances of the **myEvent** event.

4.6.3 Multiple Simultaneous Event Notifications [63] [24]

Events can have only one pending notification, and retain no "memory" of past notifications. Multiple notifications to the same event, without an intermediate trigger are resolved according to the following rule:

- An earlier notification will always override one scheduled to occur later
- An immediate notification is always earlier than any delta-cycle delayed or timed notification.

Note that according to these rules, a potential non-determinism exists. Assume that processes **A** and **B** are ready to run in the same delta-cycle. Process **A** issues an immediate notification on an event, and process **B** issues a delta-cycle delayed notification on the same event. Process **C** is also sensitive to the event. According to the scheduler semantics, processes **A** and **B** execute in an unspecified order.

Table III : Event non-determinism

Process A {	Process B {	Process C {
…	…	…
my_event.Notify();	my_event.Notify(0);	Wait(my_event)
…	…	…
}	}	}

If process **A** executes first then the event is triggered immediately, causing process **C** to be executed in the same delta-cycle. Then process **B** is executed and since the event was triggered immediately, there is no conflict and the second notification is accepted causing process **C** to be executed again in the next delta-cycle.

If, however, process **B** executes first, then the delta-cycle delayed notification is scheduled first. Then, process **A** executes and the immediate notification overrides the delta-cycle delayed notification, causing process **C** to be executed only once, in the current delta-cycle.

4.6.4 Cancelling Event Notifications

A pending delayed event notification may be cancelled using the *Cancel* method . Immediate event notifications cannot be cancelled since their effect occurs immediately.

4.6.5 Events, Signals and Clocks

In ESys.Net, all predefined signals are owners of two events: **sensitive** and **transaction**.

The **sensitive** event is triggered when the value of the signal changes from it precedent value and transaction is triggered when a value is written to the signal.

Clock objects are owners of three events: **posedge**, **negedge** and **sensitive**. (Note clocks are channels even if they appear to be signals)

The **posedge** event is triggered on the positive edge transition of the clock, **negedge** is triggered on the negative edge transition of the clock and **sensitive** is triggered on both of the formers.

4.7 Channels

With the model elements that we have seen so far - modules, signals, events... etc- it is possible to create fairly complex systems. However, like most modern system description languages, we have added the semantics of channels to the environment

[59] . Channels are the basic modeling elements for complex inter-module communication. Through interfaces that channels implement, modules may communicate with each other. A channel may abstract a simple point-to-point communication but can also model very complex Network On Chip (NOC).

Channels may be seen as modeling elements that are a cross between modules and signals.

Channels differ from signals in that they may contain structure such as sub-modules, sub-channels, and processes. They differ from modules because they implement, like signals, the ***IDeltaUpdatable*** interface (this concept will be explained later) that permits them to be synchronized with the simulation delta cycles. In this way, we have simplified SystemC's channel semantics by unifying the concept of primitive channels and hierarchical channels [63] .

Another important aspect of channels is that, since they abstract communication between modules, they permit the refinement of a model's communication elements without the modification of the communicating elements.

4.7.1 Channel Declaration and Instancing

There are two parts involved in the usage of a user defined channel:

1. The declaration of the channel
2. The instantiation of the channel in the context of a system.

Like user defined modules, all user defined channels inherit from an abstract based class called *BaseChannel*. Since the *BaseChannel* class inherits from the *BaseModule* class, channels may be constructed in the same way as modules by using sub-components, processes and interfaces.

```
1. public class MyChannel: BaseChannel, MyInterface{
2. \\ declared interface and inner components
3. public MyChannel(){}
4. public MyChannel(string name): base(name){}
5. \\ declared inter workings}
```

Code example 36: Channel declaration

If the constructor on *line 3* is used when instantiating the user defined module, a default identification name will be generated for the module instance. If the constructor on *line 4* is used, the programmer must supply the channel identification name when instantiating the channel. All channel instances identification names in the context of a module hierarchy level must be unique with all other modules and channels names.

On *line 1* we have declared that the channel implements a user defined interface called *MyInterface*.

Channels are instantiated in the same way as modules.

```
1. MyChannel channel_a = new MyChannel("channela");
2. MyChannel channel_b = new MyChannel();
```

Code example 37: Channel instantiation

The code on line 1 initializes a reference variable called **channel_a** by instantiating an object of type *MyChannel* and assigns the instance identification name "channela" to **channel_a**. The code on line 2 initializes a reference variable called **channel_b** by instantiating an object of type *MyChannel* and assigns a generated instance identification name to **channel_b**.

4.7.2 Channels and Software Interfaces

Since channels are specialized forms of user defined modules, they also have an interface that is constituted of ports. However, channels usually implement software interfaces, enabling modules to communicate with the channels through well defined method calls that are declared in the implemented interfaces. Modules are also able to declare ports with theses software interfaces.

The abstraction brought by interfaces is not really necessary, but it permits a clean separation between the communication elements of a model and the processing elements, because the implementing element hiding behind an interface may be trivially changed without changing the elements using the interfaces. Also, since we use the predefined concept of software interface's in .Net/C#, channels may

implement multiple interfaces and an interface may be implemented in different ways by different channels.

4.7.3 Sensitivity

Channels like modules may contain events, but these events may be declared with a public accessibility and used in the ***EventList*** of a process. This enables channels to be regarded as signals but with much more complicated inner-workings.

4.7.4 Channel hierarchies and inner-workings

As with module hierarchies, channels hierarchies are often useful to manage the complexity of modern communication channel modeling, since the concept of a channel is a sub-concept of a module. The inner structure and workings of a channel are designed in the same way as a module. Therefore a channel may contain sub-channels, sub-modules, event, signals, ports, processes etc.

4.7.5 The IDeltaUpdatable Interface

The ***IDeltaUpdatable*** interface is a custom interface that is defined in the ESys.Net framework. It is through this interface that elements such as signals are synchronized with the delta cycles of a simulation. The interface definition is as follows:

```
1. public interface IDeltaUpdatable {
2.   void Update();
3.   void RequestUpdate(){}}
```

Code example 38: IDeltaUpdatable interface

The *RequestUpdate* method must implement the request of a delta-cycle synchronization for the element that is making the call. The *BaseChannel* class has a default implementation for this method that calls the simulation kernel and puts the requesting element in the simulation kernel's list of elements to be updated before the next delta-cycle. The *Update* method is the method that is called to perform the delta-cycle synchronisation. It is with this method that the signals of the environment change their current value with the value that has been written to the signal during the previous delta-cycle. The *BaseChannel* class implements this method with an empty virtual method which can be overridden by subclasses.

In order to use a channel in a simple way, one can put aside the delta-cycle synchronization elements of a channel. If the delta-cycle behaviour is important, however, one must only redefine the Update method to implement the correct synchronization behaviour and call the predefined *RequestUpdate* method appropriately.

We should point out at this point that any user defined model element may implement this interface and use the simulation kernel's delta element handling method to synchronize with delta-cycles:

```
1. public void RequestUpdate(IDeltaUpdatable up )
```

Code example 39: RequestUpdate method

4.7.6 Unification of the Channel Concept [63] [24]

SystemC has two kinds of channels: primitive channels and hierarchical channels. Primitive channels are flat elements; they do not have any hierarchical structure and do not contain processes. They support, however, the synchronization of their state with the simulation kernel's delta-cycle. Hierarchical channels are modules hiding behind a "typedef", so they can have structure and processes but they are not inherently capable of being synchronized with the delta-cycles like primitive channels.

ESys.Net has unified these two entities within the concept of the *BaseChannel*. A primitive channel is just a specific case of a user defined channel that is not hierarchal, does not contain any processes and which has a custom defined *Update()* method. Also, our *BaseChannel* is truly a specialization of a module and not just a hiding alias which permits a better analysis of a model because the concepts are well defined.

4.7.7 Example [65] [24]

In Annex A we present an example that illustrates a communication channel that is designed like a SystemC primitive channel; the channel is a FIFO. This FIFO channel comes with a number of methods. Basically, we find both blocking and non-blocking I/O as well as some functions to query the state of the FIFO. In its implementation,

we use the "Request/Update" scheme. We also find a good example of dynamic sensitivity in order to implement the blocking I/O. The FIFO's read and write interfaces are given first, then the FIFO ,and then a simple example with a producer and consumer using the channel. This example is an ESys.Net partial implementation of an example in the SystemC 2.0 functional specification that is explained in depth .

Chapter 5 Simulation Kernel

The ESys.Net modeling constructs that we have seen so far are almost identical to the ones found in SystemC. This is no coincidence because, as mentioned earlier, ESys.Net is meant to be an evolution of SystemC. The true difference between the two environments lies within the implementation of their respective simulation kernels. ESys.Net innovates on SystemC by leveraging the advance software capabilities of the .Net Framework, through the use of C#, in order to create a simpler and more flexible simulation kernel. The programming features that are the foundation of the kernel are:

- attribute programming
- reflectivity
- native .Net threads
- native .Net synchronization primitives
- delegates and events

The simulation kernel is a very important aspect of our environment because it is at its heart. The kernel has many important functions such as:

- Simulation model elaboration
- Process scheduling
- Delta-cycle synchronisation
- Model correctness verification
- Mediator for third-party tools

5.1 Modeling Directives

One of the important characteristics of ESys.Net is that it offers the system designer the possibility to easily specify execution directives by tagging the different concepts in the specification. These directives concern (i) the association of a process or parallel method semantic to a class method, (ii) the addition of a sensitivity list, (iii) the calling of methods before or after the execution of a certain process and (iv) the execution of a class method at a specific moment during the execution. This was implemented by exploiting the attribute programming paradigm provided by .NET and the C# language. Table IV summarizes the available attributes, their semantics and the concepts to which they are applied.

Table IV : Attributes and their role in ESys.Net

Attribute	Description		Applied to concept
Process	Associate a thread to a class method		Class Method
PMethod	Associate a method process to a class method		Class Method
EventList (list of events)	Add sensitive list for a process		Process
ManualRegistration	Manual registration of the element		Field
PreCall (Name of Method)	Directives indicating methods execution in explicit points of the execution flow	Method to be called before the process	Process
PostCall (Name of Method)		Method to be called after the process	Process
SimInit (Name of Method)		Simulation init	Class Method
SimEnd (Name of Method)		Simulation end	Class Method
CycleInit (Name of Method)		Cycle initialization	Class Method
CycleEnd (Name of Method)		Cycle end	Class Method
DeltaInit		Delta cycle initialization	Class Method
DeltaEnd		Delta end	Class Method
FinalDelta		Last delta	Class Method
Reset		Simulator reset	Class Method

The utilisation of attribute programming offered us a very powerful tool, one that enabled us to create a transparent means to permit the addition of hardware semantics to behavioural code in a simple and elegant way.

In order to offer a declarative mechanism to add hardware semantics to a model, SystemC uses macro. We believe that the use of attributes is much more elegant than macro, because they do not hide code that must be debugged when working on a model [50] .

5.2 Simulation Semantics

Most of the differences between SystemC and ESys.Net are within the simulation kernel so both will be described and compared.

5.2.1 SystemC [64] [63]

SystemC has at its core an event based simulation kernel like most current simulation environments i.e. Verilog, VHDL. A SystemC simulation execution may be broken up in a number of consecutive phases: elaboration, initialization and process scheduling.

Elaboration phase

It is during this phase that the simulation model is created from the model description. Structural elements of the systems i.e. modules, channels, signals etc. are created and connected throughout the system hierarchy. Hierarchical structures are elaborated through recursive construction using object construction behaviour. During the elaboration phase, the simulation kernel must create a process object for each threaded and method based process.

SystemC's elaboration phase can be further broken up into two sub-phases that occur at different instances. The first sub-phase occurs at compilation time. During this sub-phase, macros are expanded revealing the code that creates the necessary process objects and retrieve pointers to the methods that are declared to become either threaded or method based processes. The second sub-phase is done during execution time. It is during this sub-phase that the structural elements are created and connected. Certain aspects of the simulation may be configured at this point.

Initialization

Initialization is the first step in the SystemC scheduler. Each method process is executed once during initialization and each threaded process is executed until a wait statement is encountered.

Process Scheduling

The SystemC scheduler controls the timing and order of process execution, handles event notifications and manages updates to channels. It supports the notion of delta-cycles. A delta-cycle consists of the execution of an evaluation and update phase. There may be a variable number of delta-cycles for every simulation time.

SystemC's processes are non-preemptive. This means that for thread processes, code delimited by two wait statements will execute without any other process interruption and a method process completes its execution without interruption by another process.

The semantics of the SystemC simulation scheduler is defined by the following eight steps. A delta-cycle consists of steps 2 through 4.

1) *Initialization Phase.*

2) *Evaluation Phase.* From the set of processes that are ready to run, select a process and resume its execution. The order in which processes are selected for execution from the set of processes that are ready to run is unspecified. The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run during the same evaluation phase. The execution of a process may include calls to the request_update() function which schedules pending calls to update() function in the update phase. The request_update() function may only be called within member functions of a primitive channel.

3) Repeat Step 2 for any other processes that are ready to run.

4) *Update Phase.* Execute any pending calls to update() from calls to the request_update() function executed in the evaluate phase.

5) If there are pending delta-delay notifications, determine which processes are ready to run and go to step 2.

6) If there are no more timed event notifications, the simulation is finished.

7) Else, advance the current simulation time to the time of the earliest (next) pending timed event notification.

8) Determine which processes become ready to run due to the events that have pending notifications at the current time. Go to step 2.

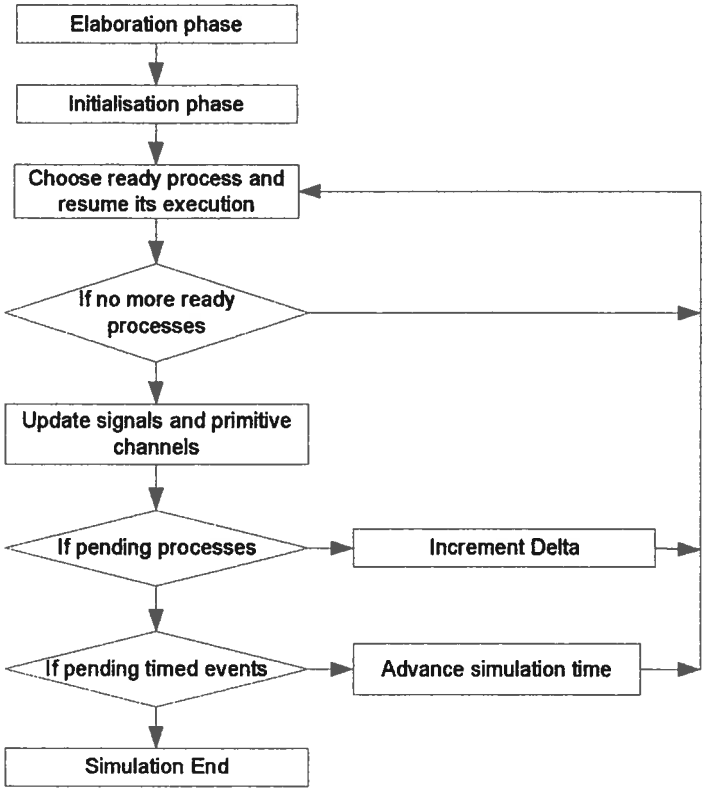

Figure 9 : SystemC's scheduler structure

Figure 9 is an overview of the simulation semantics of SystemC. Note that it is very closed; there are no entry points for third-party tools **Erreur ! Source du renvoi introuvable.**.

5.2.2 ESys.Net

This section describes elaboration, initialization and the simulation semantics. ESys.Net is an event based simulator. ESys.Net's simulation execution may be broken up into the same steps as SystemC.

Elaboration

As is the case in SystemC and most environments, it is during this phase that ESys.Net model element instances are created and connected together such as module, channels, signals etc. However, unlike must environments our elaboration phase is done dynamically at run-time; there is no code added at compile time as in SystemC. ESys.Net models do not take for granted a specific simulator. At runtime a model is bound to a simulator, that in turn through .Net's introspections capabilities, analyses the model (structure and directives) and creates a simulation representation of the model. This permits our models to be compiled separately from a specific simulator and to bind the models at a later time to a specific simulator. This also allows us to have many simulator works on different models or parts of models in a unique binary execution. Verilog, VHDL and SystemVerilog take for granted that there is only one simulator so it is implicit and it is during compilation that a model is bound to it (there can only be one model per simulation).

The first part of the elaboration phase is to discover and register the various elements of the model; code example 40 presents an incomplete pseudo-code of the algorithm we use to do these tasks.

```
1. SubModelElementRegistration(ModelObject element)
2. type := GetType(element)
3. fields:= GetAllFields(type)
4. foreach field in fields
5.   if Not(ManualRegisteredTagged(field))
6. field instance:= GetFieldInstance(field,element)
7. select(field instance)
8.   case Clock:
9.         RegisterClock(field instance)
10.  case Channel:
11.        RegisterChannel(field instance)
12.        SubModelElementRegistration(field instance)
13.  case Module:
14.        RegisterModule(field instance)
15.        SubModelElementRegistration(field instance)
16.  case Signal:
17.        RegisterSignal(field instance)
18.  case Event:
```

```
19.        RegisterEvent(field instance)
```

Code example 40: Model Discovery and Registration

Each object in the model is passed through this algorithm. Each field in the currently treated object (element) is extracted and registered in accordance to its base type i.e. module, channel, clock, event or signal. If a field is module or channel, it is recursively passed through the same algorithm. However, if a field is tagged with the ***ManualRegistered*** attribute it is not processed.

The second part of the elaboration phase is the heart of the kernel. The algorithm that we use creates the simulation model. The algorithm is responsible for the creation of the processes and for the binding of these processes to their triggering static events. The algorithm also binds class methods to hooking points within the kernel.

The simulation model construction algorithm can be broken up into four parts:

- Process discovery and verification
- Process method creation
- Parallel Method creation
- Callback hooking

Each part will be presented with an incomplete pseudo-code.

```
1. SimulationModelCreation()
2. For each element in RegisteredChannels and RegisteredModules
3.   type:=GetType(element)
4.   if type is an Interface
5.         type:= GetDeclaringType(type)
6.   methods:=GetAllPrivateMethods(type)
7.   foreach method in methods
8.      parameters:= GetParameters(method)
9.      if Size(parameters)=0
10.        method instance:= GetMethodInstance(method,element)
11.        select(method)
12.              case ProcessTagged:
13.              …
14.              case PMethodTagged:
15.              …
16.              case Default:
17.              …
```

Code example 41: Process dicovery and verification

Code example 41 gives the pseudo-code that goes through all the registered channels and modules, and retrieves their method declarations that have a private scope. For each method declaration found, the algorithm verifies if the method is eligible to be a process (e.g. verifies if the method takes no arguments and returns void). The code then gets the method instance for the currently processed object and then verifies if the method declaration has either a ***Process*** tag, a ***Pmethod*** tag or callbacks tags.

```
1. thread:= CreateThread(method instance)
2. process:= CreateProcess(thread)
3. RegisterProcess(process)
4. if HasEventList(method)
5.   event list: = GetEventList(method)
6.   owners:= GetOwners(event list)
7.   event name:= GetEventName(event list)
8.   field instances:= GetOwnersFromElement(element)
9.   for each field instance in field instances
10.        event:= GetEvent(event name, field instance)
11.        Bind(event, process)
12. if Not(HasEventList(method)) or SimInitTagged(method)
13.  ExecuteAtSimulationInitialisation(process)
14. if PreCallTagged(method)
15.  premethods:= GetPreCallMethods(method)
16.  foreach premethod in premethods
17.     premethod instance:= GetMethodInstance(premethod, element)
18.     PreCallHook(process, premethod instance)
19. if PostCallTagged(method)
20.   postmethods:= GetPostCallMethods(method)
21.   foreach postmethod in postmethods
22.   postmethod instance:= GetMethodInstance(premethod, element)
23.   PostCallHook(process, postmethod instance)
```

Code example 42: Algorithm part for process methods

The block of pseudo-code in example 42 is responsible for the creation and management of process methods. It creates a thread for the method instance and then creates a process method object for the thread. The process method object is then registered in the kernel. Processing of the static event list is done next. If no static event list is declared, the process method is registered to be executed before the simulation starts. Lines 5 to 11 process the process method's event list and binds it to the correct event objects. The rest of the code deals with methods that must be called

before or after the execution of the process method; these methods are good for pre and post condition verification.

```
1. pmethod := CreateProcess(method instance)
2. RegisterProcess(pmethod)
3. if HasEventList(method)
4.   event list: = GetEventList(method)
5.   owners:= GetOwners(event list)
6.   event name:= GetEventName(event list)
7.   field instances:= GetOwnersFromElement(element)
8.   for each field instance in field instances
9.     event:= GetEvent(event name, field instance)
10.    Bind(event, pmethod)
11. if Not(HasEventList(method)) or SimInitTagged(method)
12.    ExecuteAtSimulationInitialisation(pmethod)
13. if PreCallTagged(method)
14.    premethods:= GetPreCallMethods(method)
15.    foreach premethod in premethods
16.      premethod instance:= GetMethodInstance(premethod, element)
17.      PreCallHook(pmethod, premethod instance)
18. if PostCallTagged(method)
19.    postmethods:= GetPostCallMethods(method)
20.    foreach postmethod in postmethods
21.      postmethod instance:= GetMethodInstance(premethod, element)
22.      PostCallHook(pmethod, postmethod instance)
```

Code example 43: Algorithm part for parallel methods

The pseudo-code in example 43 does almost the same thing as the previous example but no thread is created for the parallel method.

```
1. if SimInitTagged(method)
2.       SimInitHook(method instance)
3. if SimEndTagged(method)
4.       SimEndHook(method instance)
5. if CycleInitTagged(method)
6.       CycleInitHook(method instance)
7. if CycleEndTagged(method)
8.       CycleEndHook(method instance)
9. if DeltaInitTagged(method)
10.      DeltaInitHook(method instance)
11. if DeltaEndTagged(method)
12.      DeltaEndHook(method instance)
13. if LastDeltaTagged(method)
14.      LastDeltaHook(method instance)
```

```
15. if PreDeltaUpdateTagged(method)
16.     PreDeltaUpdateHook(method instance)
17. if PreDeltaIncTagged(method)
18.     PreDeltaIncHook(method instance)
```

Code example 44: Algorithm part for callback hooking

The code block in example 44 manages the binding of methods to the various callback points in the simulation kernel.

Initialization

Initialization is the first step in the ESys.Net scheduler. Processes are not executed by default, only processes that have been tagged with a ***SimInit*** directive or processes that don't have a sensitivity list (transaction level processes) are executed during this phase.

Process scheduling

The ESys.Net scheduler controls the timing and order of process execution, handles event notifications and manages updates to channels and signals. It supports the notion of delta-cycles. ESys.Net processes are pre-emptive. The semantics of the ESys.Net simulation scheduler are defined by the following eight steps. A delta-cycle consists of steps 3 through 11. As illustrated by the steps of a simulator scheduler, we have added many hooking points (steps in bold) within our simulation kernel.

1) *Initialization Phase.*
2) **Execute cycle initialization callbacks**
3) **Execute delta initialization callbacks**
4) *Evaluation Phase.* From the set of processes that are ready to run, select a process. The order in which processes are selected for execution from the set of processes that are ready to run is unspecified.
5) **Execute pre-method callbacks for the current process**
6) Resume current process's execution
7) **Execute post-method callbacks for the current process**
8) Process execution (The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run in the same evaluation phase)

9) Repeat Step 4 for any other processes that are ready to run.

10) Execute pre-update callbacks

11) *Update Phase*. Update delta-cycle dependent elements that requested updates (signals and primitive channels)

12) If there are pending delta-delay notifications, determine which processes are ready to run and go to step 3.

13) Execute last delta callbacks

14) If there are no more timed event notifications, go to step 18.

15) Else, execute cycle end callbacks

16) Advance the current simulation time to the time of the earliest (next) pending timed event notification.

17) Determine which processes become ready to run due to the events that have pending notifications at the current time. Go to step 2.

18) Execute simulation end callbacks

19) The simulation ends here.

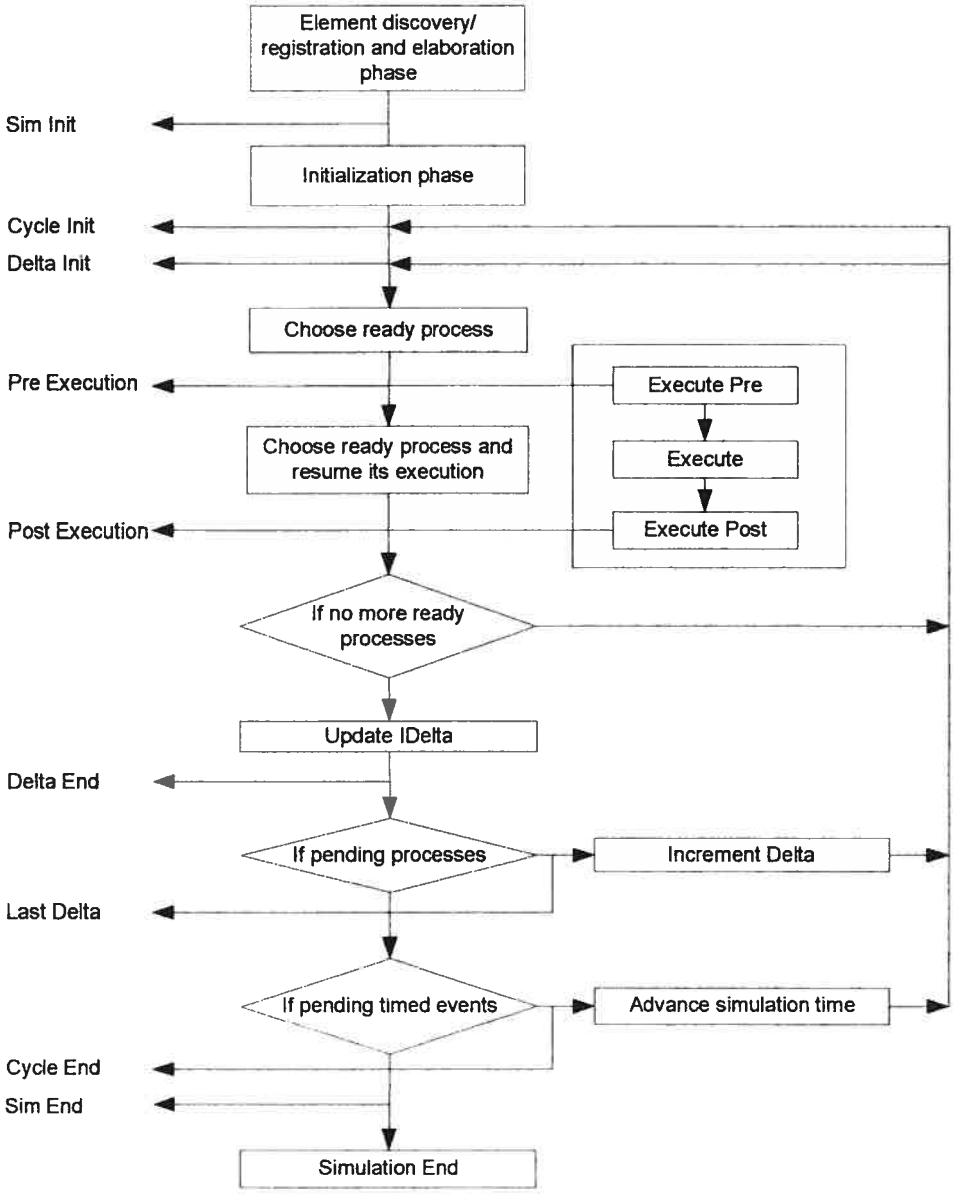

Figure 10 : ESys.Net Scheduler Steps

Figure 10 gives a good view of the simulation kernel. Compared to SystemC, our simulation kernel has many callbacks points for third-party tool binding.

Since our hooking points are implemented with delegates and events, it is possible to hook many callbacks to a same entry point and callbacks may be class instance methods or static class methods. Also, since we are using delegates, it is possible to bind a method to a hook point at runtime because it is not necessary to know at compile time the names of the methods we want to bind to a delegate.

The following is a simple code example that illustrates the instantiation of a model, a simulator, and a verification tool, and the binding of the elements. Notice that with a simple line of code we can bind a method to the ***CycleInit*** event of a simulator. The ***CycleInit*** event is triggered at the beginning of every simulation cycle.

```
1.  public class SimpleBusApp{
2.   static public void Main(){
3.   My VeriTool tool = new VeriTool();
4.   MyModel  model = new MyModel(sim);
5.   Simulator  sim = new Simulator(sbt);
6.   sim.cycleInit += new HookingPoint(tool.verifie);
7.   sim.Run(100000);}}
```

Code example 45: Tool hooking

Chapter 6 Comparison and Experimental Results

ESys.Net is not meant to be a new solution but rather an evolution of an existing solution: SystemC. For this reason we will not discuss the pros and cons of pure modeling semantics and library based approach because SystemC has already addressed theses points. Rather, we will discuss here the pros and cons of our design compared to those of 'SystemC'. Also since ESys.Net is intended to be an evolution to SystemC, we shall not compare it to other environments because many articles already exist that compare SystemC to other alternatives [68] [8] [9] .

We will present some experimental results collected on performance issues and an example that illustrates the ease of connection of a third-party tools to the simulation kernel.

6.1 Advantages of the Environment

ESys.Net design has many advantages over those of SystemC. The advantages can be categorized into three aspects: the simplification of semantics and programming, and a more open integration.

6.1.1 Semantic Simplification

ESys.Net has simplified SystemC's more advanced modeling concepts such as ports and channels.

Unification of ports and interfaces

As mentioned in a previous section, we have unified the concept of ports and interfaces. This unification has been done for several reasons. Firstly, in SystemC, ports are intermediate objects that have a predefined interface on which processes make calls. On a method call, a port delegates the call to another interface, which hides a channel that is contained within it. The predefined interfaces that ports support in SystemC are very basic – simple read and write calls; for this reason, a user must use indirection on a port to get a pointer on the contained interface and then make an indirect call to that interface. Since SystemC's main objective is system level modeling, very few designs will be able to use simple read/write calls that do not take any arguments to model complex buses like the ones used in NoC. SystemC has even questioned the usefulness of ports, but decided to keep them for static binding verification purposes.

> "The question arises whether port objects are needed at all – one could argue that it would be sufficient to only have the notion of interfaces being implemented by channels. Basically, port objects serve a dual purpose. Firstly, they allow for the implementation and enforcement of (static) design rule checks. Secondly, they provide objects that can be attached attributes such as names or priorities."[24]

The first argument based on design rule verification is not so justified with .Net/C#. SystemC provides a mechanism that allows channels to verify static design rules when ports are bounded to the channel. The mechanism is provided by a virtual method called register_port() that channels may override. The method is called when a port is bound to the channel. Because SystemC is based on C++, this mechanism would be very hard to implement without ports. ESys.Net does not currently have anything equivalent to this verification mechanism, but an equivalent mechanism could be fairly easily created. Since ESys.Net is based on .Net/C#, reflective capabilities could be used to analyze a model after the elaboration phase and fill a data structure in the individual channels with information such as: what modules have a reference to the channel, which interface is used in the module to abstract the channel etc.

The second argument supporting the usefulness of ports for attaching extra information such as priorities and name is not valid with the help of attribute programming. Metadata may be attached directly to the interface variable declaration:

```
1. public class FullAdder: BaseModule{
2. [Priority(1)]public inBool a;
3. public inBool b;
4. …
```

Code example 46: Metadata (priority)

By unifying the concept of a port and an interface, we have simplified our design library. Moreover, our models take less memory because there is no memory allocated for an interface.

Unification of primitive and hierarchical channels [24]

SystemC has divided the semantics of high level communication into two orthogonal entities: primitive channels and hierarchical channels. The concept of hierarchical channels is not truly defined because they are just modules hiding behind a "typedef". We find that this separation is quite arbitrary and that hierarchical channels should be better defined.

ESys.Net unifies the two entities making the concept of primitive channels a specific case of the general concepts of hierarchical channels. This simplification makes for a simpler environment. Moreover, the concept of a channel in ESys.Net is clearly separate from the concepts of modules. This separation will permit tools to easily distinguish channels from modules in the same way our discovery algorithm does.

6.1.2 Programming Simplification

The main purpose of system design languages is to model and simulate complex systems at high and low levels of abstraction. At high levels of abstraction, systems should be easily modelled and quickly debugged. Since SystemC is based on C++, it is plagued with all the complexities of the languages such as pointers, manual memory management and macros just to name a few. Because of these programming complexities, even simple systems become overly complex to model (just dealing

with header files can be a headache). System designers should be good at designing systems not necessary programming with C++ [17] .

The C# programming language offers a simple and elegant basis for the ESys.Net environment. With C#, designers can focus on the modeling of systems instead of the ins and outs of the supporting language. Moreover, the dynamic verification that the .Net runtime does on executing code permits the creation of less error prone systems.

6.1.3 A Simpler Better Framework

Taken as is and for what it is, SystemC is a very effective environment. However, because SystemC is based on a fairly low level language like C++, its evolution and customization is greatly hindered by its underlining design. For performance reasons and because C++ does not support reflective capabilities, SystemC makes excessive use of macros and obscure design techniques. To stay cross platform, SystemC makes use of a user side thread library. This is not a problem as such, but when debugging custom modifications to the simulation kernel, the debugger must go through this complex code which is a problem. Moreover, custom modifications to the kernel are often necessary because SystemC's design was not intended for hooking third-party tools to the simulation kernel [53] [11] . ESys.Net alleviates all these problems by leveraging the many features of .Net/C#.

Simpler design

By leveraging the already rich runtime and class framework provided by .Net/C#, ESys.Net has a much smaller and simpler design that SystemC. By using the threading capabilities provided with the runtime, ESys.Net offers cross-platform threads whose implementation is hidden from the user. This simplifies the debugging of the simulation kernel. It also shortened the time necessary to develop the kernel because the thread library is almost trivial compared to QuickThreads (the library used by SystemC).

By using attribute programming reflectivity and delegates, the use of pointers, macros and "typedef's" were completely eliminated, offering a simple design that is easy to

debug. ESys.Net is simpler to understand and debug because there are no macro expansions at compile time, so what is debugged is what is written and not what has been expanded. Pointers are also a problem to debug, especially when they are function pointers.

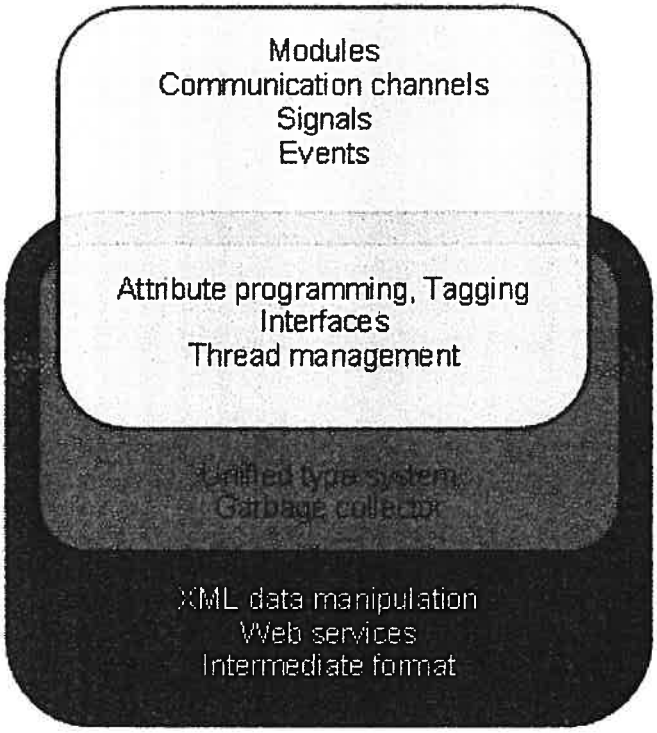

Figure 11 : Leveraging of existing features

Figure 11 shows how ESys.Net was designed with a layered approach by using already existing features found in .Net and C#. The further box represents the scope of .Net, the middle box represents the scope of C# and the top box represents the scope of ESys.Net.

When features are found in overlapping boxes, it means that the features are defined in the lower box but are leveraged by the higher ones. We can see by the diagram that ESys.Net only needed to implement the missing hardware semantics such as modules signals, etc.; all the rest comes from .Net and C#.

As an example of how much code was saved by using .Net and C# to implement ESys.Net, here is a comparison of the scheduling kernels: SystemC's kernel is very large, just the scheduling kernel is well over 1500 lines of code (this is not counting

the lines of code used by the user side thread library which is about the same size and the code that is hidden by macro expansions), ESys.Net is only about 500 lines of code. Because we leverage the use of an advanced programming environment, the ESys.Net kernel is very simple and can almost be read like pseudo-code which is not the case with SystemC.

Open design

The business and academic worlds believe we are going through a design crisis because the computation power of hardware technologies is growing exponentially but our ability to produce effective design for these technologies is not [27] . Environments such as SystemC help designers build and test complete systems rapidly, but design exploration and verification through simulation has its limits. In order to get past the current design crisis, new sophisticated CAD and EDA tools are required to perform advanced design analysis and verification. CAD and EDA tools could also automate certain aspects of design space exploration with the use of constraints.

Some environments such as Verilog and SystemVerilog have well defined APIs that permit the hooking of third-party tools to the simulation kernel [32] [54] [4] . These APIs usually permit the registration of callback functions or the ability to introspect the current model being simulated. However, most of the APIs are overly complex limiting the rapid design of custom tools. SystemC, because of its design, does not easily support the hooking of third-party tools. Many have had to modify the kernel[11] [53] . Also, since C++ offers no standard reflective capabilities, model introspection is very limited even with the new SystemC Verification Library.

ESys.Net makes third party tool hooking simple by providing numerous callback points in the simulation kernel, making kernel modification almost avoidable in most cases. Through the use of .Net/C# reflective capabilities, it is feasible for small development teams to design sophisticated analysis tools rapidly.

The following code examples are excerpts of a complete programme in Annex C. The program demonstrates the powerful reflective capabilities of .Net/C# and the simple

elegance of third-party tool hooking with ESys.Net. The program is a complete implementation of the simple "MyFirstSystem" example that we presented in section 4.1. The program also contains the code for a verification tool that we will be presented briefly below. The verification tool, once bound to a system model, can discover all the signals and ports contained within the model. It then keeps a list of the elements it finds. The verification tool is then capable of printing out the current value and name of each signal and port. In the code excerpts that follow, we will rapidly present the algorithms used for the discovery and printing as well as the code that binds the tool to the simulator.

```
1. private void DiscoverSignals(ImodelElementContainer
2.           element,string parentname) {
3.   Type type = element.GetType();
4.   ArrayList hierarchialElements = new ArrayList();
5.   Object[] couple;
6.   foreach(FieldInfo fi in type.GetFields(BindingFlags.Instance|
7.           BindingFlags.Public  |
8.           BindingFlags.NonPublic)){
9.    Object obj = fi.GetValue(element);
10.   if(obj is BaseModule){
11.     couple = new Object[2];
12.     couple[0] = parentname +  "." + fi.Name;
13.     couple[1] = obj;
14.     hierarchialElements.Add(couple);
15.   }else if(obj is BaseSignal){
16.     couple = new Object[2];
17.     couple[0] = parentname +  "." + fi.Name;
18.     couple[1] = obj;
19.     signals.Add(couple);
20.   }
21. }
22. foreach(Object[] pair in hierarchialElements)
23.   DiscoverSignals((IModelElementContainer)pair[1],
24.       (string)pair[0]);
25.}
```

Code example 47: Signal Discovery method

Code example 47 is the code fragment we use to discover the signals and ports of the system model. It is based on the same pseudo-code explained in Chapt.5 that the simulation kernel uses to discovers the various elements of a model.

```
1. public void PrintSignals(){
2.   foreach(Object[] pair in signals){
3.     Object currentValue = pair[1].GetType().
4.     GetProperty("Value").GetValue(pair[1],null);
5.     Console.WriteLine("----------------------------");
6.     Console.WriteLine("Signal/Port name: {0};",pair[0]);
7.     Console.WriteLine("Current value: {0};",currentValue);
8.     Console.WriteLine("----------------------------");
9.   }
10.}
```

Code example 48: Printing method

Code example 48 gives the algorithm used to print the current value of the signals and ports found by the tool, as well as their hierarchical names.

```
1. static void Main(string[] args) {
2.   Simulator sim = new Simulator();
3.   MyFirstSystem sys = new MyFirstSystem(sim);
4.   SignalPrinter sp = new SignalPrinter(sys,"MyFirstSystem");
5.   sim.simInit+= new RunnableMethod(sp.Initialize);
6.   sim.cycleInit+= new RunnableMethod(sp.PrintSignals);
7.   sim.Run(20);
8. }
```

Code example 49: Tool hooking

Code example 49 is responsible for hooking the tool's initialization method to the ***SimInit*** event of the simulator as well as the tool's printing method to the ***CycleInit*** event. When the simulator starts its initialization phase, it will cause the tool to initialize itself. At the beginning of each simulation cycle, the tools will print the value and names of the signals and ports. The following is part of the simulation output (the complete output is in Annex B)

Excerpt of the ouput:

```
----------------------------
Signal/Port name: MyFirstSystem.sig3;
Current value: 0;
----------------------------
----------------------------
Signal/Port name: MyFirstSystem.top.porta;
Current value: 0;
----------------------------
```

With SystemC, a simple but effective tool like the one presented in the previous pages, is unreasonably complex to create. The simple elegance of third party tools hooking that ESys.Net supports, we believe, is in itself a major contribution to the community.

6.2 Disadvantages of the Environment

One advantage of SystemC is that people can model software parts of the overall system and then simulate them. Once those parts are verified, they can be compiled with already existing tools for a vast number of processors because C/C++ are still the major languages used for software development (especially embedded systems). C# currently lacks this capability because it relies on a runtime.

The generic programming features of C++ don't have any equivalent in C#. It is possible to simulate generic programming by writing algorithm with only variables of type Object. This does not permit the compiler to do static type check and it also forces the user to use a lot of type casting (slowing down simulation). Generic programming would simplify the creation of custom user signals that are type safe and faster to execute. It would also simplify the ESys.Net library because we had to create a signal for each basic value type supported by the CTS. This said, generic programming features have been announced for the next version of .Net\C#; there is already an experimental version that is available called Gyro [37] .

Another advantage of SystemC is that its simulation library is more complete. It contains elements for the modeling of fixed point floating types, some predefined channels and the simulator supports many time units just to name a few. All of these features may be modeled in ESys.Net. We did not implement them because they did not add any value to our proof of concept.

6.3 Experimental Results

To prove the efficiency of ESys.Net we performed several experiments. The main criteria that we used for the evaluation were the performance and the applicability for concrete systems modeling and simulation.

Firstly, we compared the performance of the C# language to the C++ language using a concrete application, the simulation model of a DLX processor. We measured the simulation time of this application for the C# specification execution on .NET and the C++ description executing natively. The results obtained were that the two languages present comparable capabilities in terms of simulation speed – the C# execution time penalty was below 10%. These results are in concordance with reported numbers concerning the use of C# versus C for real-time applications. We consider this penalty acceptable given the advantages of the .NET framework.

In addition, we modeled and simulated a second concrete application. In order to compare with a well known simulator, we used an application provided by SystemC. The overview of this application is illustrated in Figure 12. The application consists of 7 main components (Annex A):

- a communication channel that ensures the communication between the other components of the system. These components may be masters or slaves of the channel. A master module requires communication primitives from the channel and the slave module offers services to the communication channel;
- two memories (the fast memory and the slow memory) that differ by the number of clock cycles necessary to read or write data; the memories are slave modules of the communication channel;
- three master modules that read/write date to/from the memories through the communication channel;
- an arbiter that provides a priority based management for the concurrent requests from the masters.

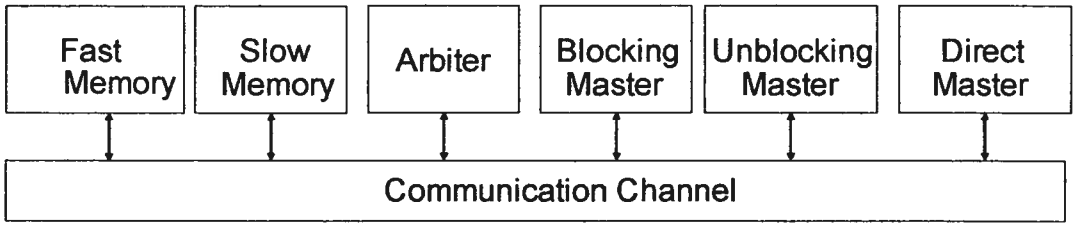

Figure 12 : General view of the application

We obtained the correct simulation of the system (verified by comparing the results in our environment with those given by SystemC).

The CLI specification does not mention if threads should be pre-emptive or collaborative. Microsoft thread implementation is pre-emptive. Pre-emptive threads permit the modeling of concurrent software components in a precise way, enabling the verification of race conditions and dead lock. SystemC lacks the capability because its threads are collaborative. To evaluate the performance penalties of context switches due to pre-emptive threads, we conducted a simple simulation that modeled the worse case scenario of a single variable affectation during a time slice. We then added progressively more computation during a time slice to see how the cost would evolve. Here is a snippet of the code:

```
1. [Process]
2. [Event List("sensitive","InA")]
3. private void Run(){
4. int i;
5. while(true){
6. for(i=0;i<LIMIT;i++){}
7. InA.Value = i;
8. Wait();
9. }
10.}
```

Code example 50: Context switch verification

We modeled this process in ESys.Net and SystemC with threaded processes and methods based processes. Figure 13 compares the execution time of ESys.Net and SystemC when the LIMIT parameter of the "for" statement is increased.

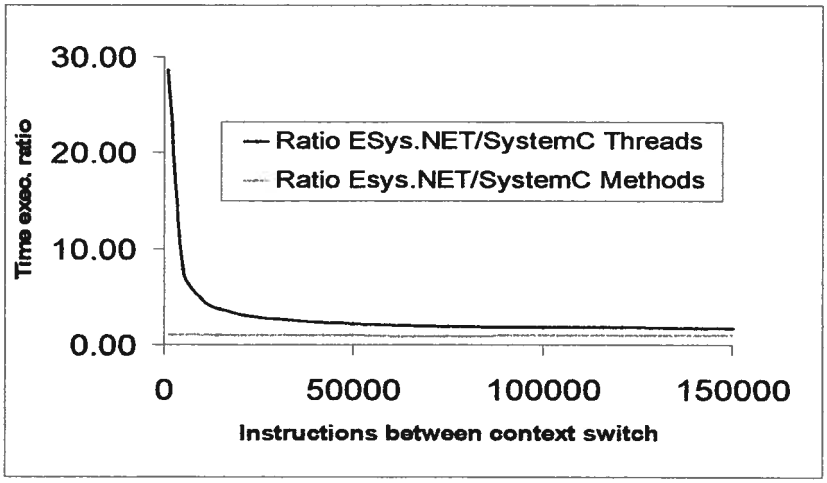

Figure 13 : ESys.Net versus SystemC performance

The experiment shows that our environment may present a performance penalty. Even if the computation in .NET threads is executed efficiently, the context switch and the thread instantiation in the current implementation of .NET are relatively costly compared to QuickThreads used by SystemC. Consequently, the penalty is reduced proportionally with the growing complexity of the computation performed in the threads. The overall performance of .NET threads compared to SystemC threads may vary from 30 times for light threads to less than 75% penalty for computation intensive .NET threads. Using method based processes results in a penalty lower that 10% [42] .

It is important to emphasize that this performance cost is compensated by two advantages that are presently missing in SystemC: the possibility of modeling real multi-threading and the possibility of modeling software components at different abstraction levels. These advantages are foreseen in the future version of SystemC and the cost to pay for introducing them have not yet been evaluated. Consequently, the comparison between ESys.Net and SystemC thread performances is quite unfair because they are conceptually different. Our experiments only quantify the penalty of having a real multi-thread based environment.

6.4 Summary

Here is a table that summarize the advantages and disadvantages of ESys.Net compared to SystemC.

Table V : Summary of the Advantages and Disadvantages over SystemC

Advantages	
Simpler programming basis	The C# programming language is a simpler language than C++ that offers many high level programming constructs, hence programmer are more productive because they can code faster and the code is less error prone. Using C# also permits designers to concentrate more on modeling tasks then on eclectic language specificities.
Semantic unification	The ESys.Net Framework has unified some of the modeling concepts found in SystemC : ports and interfaces, channels and primitives channels, which simplifies the modeling framework and makes it smaller.
Open design	Third party tool integration and CAD tool creating is made simple by the use of reflection.
Simpler design	The ESys.Net kernel is many times simpler then the SystemC kernel because it is based on the high level programming constructs of C# and the .Net Framework. Our simpler design permit the ease debugging of custom modifications.

Disadvantages	
Incomplete library	The ESys.Net Framework is not as complete as SystemC 's - we did not create a complete library because it did not add to our proof of concept. A complete library can easily be created.
Code speed	The Common Intermediate language execution is about 10% slower then compiled C++ code. However we find this cost insignificant compared to the productivity gains and runtime support offered by .Net.
Thread speed	The modeling processes that are implemented with .Net threads have a high performance penalty compared to SystemC's processes that are modeled with QuickThreads. However, by using threads that are pre-emptive and scheduled by the OS, ESys.Net permits the modeling of concurrent software tasks and may utilize the parallelisms offered by a multiprocessor system to gain simulation speedups. If the ability to use fibers to implement threads in .Net became available, the performance cost would probably be completely eliminated [57] .

We believe that the advantages of ESys.Net gain though using the .Net Framework greatly outweigh the performance penalties that are incurred. Also, we believe that it is important for the next generation of modeling and simulation tools to put aside issues of performance penalties that have a constant cost because the growth of system models is exponential – even a large constant gain in performance will rapidly become insignificant with the rapid size growth of system models. We believe that

the next generation of solution should put the emphasis on higher modeling abstractions and higher design productivity.

Chapter 7 Summary and Future Work

Nowadays, in order to respect the time to market and strict cost constraints, system designers need new modeling and simulation solutions. These solutions must enable easier memory management and software component complex specifications, multi-language features and mitigated connection with other existing or new CAD tools.

In this thesis a new solution for modeling and simulating called ESys.Net was presented. ESys.Net brings to the hardware/software modeling community a new solution that has all the benefits of SystemC without having most of its drawbacks such as: the complexity of the C++ language, the complexity of the modeling library, the lack of introspection, etc. Our solution also fulfills all the requirements that we enumerated in the introduction and with no significant performance cost. The solution that we propose in this research project is based on the advanced programming capabilities of the C# programming languages and of the .Net Framework runtime. By leveraging these capabilities, we have developed an environment called ESys.Net which is meant to be an evolution to SystemC.

Before SystemC become available, designers had to purchase very expensive proprietary environments for hardware and system modeling. SystemC is distributed free of licensing fees which permits small and medium size companies to do design work without a big investment up front. SystemC is also "open source", so companies can modify the environment to incorporate their own custom tools. However, custom tool integration with SystemC is very difficult and time consuming because of C++ and because of SystemC's design, so difficult that major proprietary modification to the kernel usually have to be made. This has pushed the development of custom tools out of the realm of reality of many companies, leaving them with the only option of

purchasing expensive "off the shelf tools" that do not always fulfill all requirements the companies would like.

Today, even though designers can use SystemC at no cost, if they want to do any complex modeling, they are almost obliged to go back to buying software with six digit price tags. By using ESys.Net, designer can avoid being slaves to expensive "off the shelf" "one size fits all solution" (most of us know that one size fits all does not exist!). ESys.Net is meant to be a "free" and "open source" solution that will allow designer to model systems quickly and effectively, and will also allow them to create sophisticated custom CAD tools cheaply.

7.1 Summary

ESys.Net offers many advantages over its predecessor. Among these are (i) a reduced set of modeling semantics due to concept unification, (ii) a simple programming basis exempt of eclectic syntactic elements, (iii) a simulation kernel that supports third-party tools integration, (iv) an overall environment that is better suited to less prone models, (v) a rich software library that permits the modeling of complex software components (especially operating system elements).In this thesis many concepts were examined and reviewed. We gave an overview of the different environments available for the modeling and simulation of hardware/software systems. These environments where described only briefly because it was not our intention to justify our solution in regards to these alternatives. We also presented a brief introduction to the new challenges facing system designers today and in the future and new software technologies that might help to solve these problems. We presented the .Net Framework and the C# programming language in order to expose their advance features which make up the backbone of ESys.Net. A large part of this thesis explains the various aspects of our environment. We also compared the simulation kernels of SystemC and ESys.Net because that is where their most significant differences lay. In the last pages, we objectively presented the pros and cons of ESys.Net vs SystemC. We gave some experimental results to justify our solution and we gave an irrefutable example to demonstrate the shear power of our environment.

7.2 Where Do You Go From Here?

The following are some of the many areas that should be investigated in order to expand on the work that was done to develop ESys.Net further:

- Behavioural synthesis
- System partitioning exploration
- Integration of linear temporal logic and assertion verification
- Heterogeneous system modeling
- Cosimulation with SystemC
- Visualisation and model analysis tools

The next important steps should be, however, the optimisation of the simulation kernel. Even though good performance results were collected, many implementation improvements could and should be done.

In conclusion we would like to point out that this research also confirms that software expertise might bring about substantial contribution to the hardware and system modeling domain.

References

[1] ASML Home Page, www.research.microsoft.com/foundations/AsmL/, 2003.

[2] Albahari, B., *"A Comparative Overview of C#"*, genamics.com/developer/csharp_comparative.htm, 2003.

[3] Bailey, S., "Comparison of VHDL, Verilog and SystemVerilog", Model Technology, Wilsonville, OR, Digital Simulation White Paper, July 2003.

[4] Bailey, S., "VHDL-200X improves design and verification", *EEDesign*, November 7 2003.

[5] Bellows, P. and Hutchings, B., "JHDL-an HDL for reconfigurable systems", *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, pp. 175-184, April 1998.

[6] Borrione, D., Piloty, R., Hill, D., Lieberherr, K.J. and Moorby, P., "Three decades of HDLs. II. Conlan through Verilog", *IEEE Design & Test of Computers*, vol. 9, issue 2, pp. 54-63, June 1992.

[7] Buchenrieder, K., Pyttel, A. and Sedlmeier, A., "A powerful system design methodology combining OCAPI and Handel-C for concept engineering", *Design, Automation and Test in Europe*, Paris, France, pp. 870-874, March 2002.

[8] Cai, L., Verma, S. and Gajski, D., "Comparison of SpecC and SystemC Languages for System Design", University of California, Irvin, Technical Report CECS-03-11, May 2003.

[9] Charest, L. and Aboulhamid, E.M., "A VHDL/SystemC Comparison in Handling Design Reuse", *International Workshop on System-on-Chip for Real-Time Applications*, Banff, Canada, pp. 79-85, July 2002.

[10] Charest, L., Aboulhamid, E.-M. and Bois, G., "Applying patterns and multi-paradigm approaches to hardware/software design and reuse", *in Patterns And Skeletons For Parallel And Distributed Computing,* F. Rabhi and S. Gorlatch, Eds. London: Springer-Verlag, pp. 297-325, 2002.

[11] Charest, L., Reid, M., Aboulhamid, E. M. and Bois, G., "A Methodology for Interfacing Open Source SystemC with a Thrid Party Software", *Design, Automation and Test in Europe*, Munich, Germany, pp. 16-20, March 2001.

[12] Chu, Y., Dietmeyer, D.L., Duley, J.R., Hill, F.J., Barbacci, M.R., Rose, C.W., Ordy, G., Johnson, B. and Roberts, M., "Three decades of HDLs. I. CDL through TI-HDL", *IEEE Design & Test of Computers*, vol. 9, issue 3, pp. 69-81, September 1992.

[13] Comparison of VHDL, Verilog and System Verilog, www.bitpipe.com/, 2003.

[14] Delpasso, M., Bogliolo, A. and Benini, L., "Virtual Simulation of Distributed IP-Based Designs", *Design Automation Conference*, New Orleans, LA, pp. 50-55, June 1999.

[15] DotGNU Home Page www.gnu.org/projects/dotgnu/, 2003.

[16] Doulos "A Brief History of VHDL" www.doulos.com/knowhow/vhdl_designers_guide/a_brief_history_of_vhdl/, 2003.

[17] Doulos, "*SystemC In Europe: Current Usage and Future Requirements*", www.doulos.com/, 2003.

[18] Drucker, L., "SystemC Verification Library speeds transaction-based verification", *EEDesign*, February 24 2003.

[19] ECMA-334:December 2002, C# Language Specification.

[20] ECMA-335:December 2002, Common Language Infrastructure (CLI)

[21] Extensible Markup Language (XML), www.w3c.org/XML, 2003.

[22] F. Doucet, S. Shukla, and R. Gupta, "Introspection in system-level language frameworks: meta-level vs. integrated", *Design, Automation and Test in Europe*, Munich, Germany, pp 382-387, March 2003.

[23] Ferrandi, F., Rendine, M. and Sciuto, D., "Functional verification for SystemC descriptions using constraint solving", *Design, Automation and Test in Europe*, Paris, France, pp. 744-751, March 2002.

[24] Functional Specification For SystemC 2 .0, www.systemc.org, 2003.

[25] Goering, R., "Accellera outlines major SystemVerilog enhancements", *EEDesign*, December 4 2003.

[26] Gough, K.J., "Stacking them up: a comparison of virtual machines", *Australisian Computer Systems Architecture Conference*, Queensland, Australia, pp. 55-61, January 2001.

[27] Hara, Y., "Researchers describe embedded processor design tool", *EEDesign*, May 9 2002.

[28] Hutchings, B. and Nelson, B., "Developing and debugging FPGA applications in hardware with JHDL", *Thirty-Third Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, pp. 554-558, vol. 1, October 1999.

[29] Hyperdictionary, "Callback Definition", www.hyperdictionary.com/computing/callback, 2003.

[30] IEEE Std 1076.1-1999:1999, IEEE standard VHDL analog and mixed-signal extensions.

[31] IEEE Std 1076-1987:1988, IEEE standard VHDL language reference manual.

[32] IEEE Std 1364-1995:1996, IEEE standard hardware description language based on the Verilog(R) hardware description language.

[33] ITRS, "*International Technology Roadmap for Semiconductors, Design*", 2001.

[34] Jerraya, A. and Ernst, R., "Multi-language system design", *Design, Automation and Test in Europe*, Munich, Germany, pp. 696-699, March 1999.

[35] Jia, H. and Liu, J., "Developing remote virtual instrument laboratory (RVIL) based on browser/server pattern", *International Conferences on Info-tech & Info-net*, Beijing, China, pp. 267-272, vol.4, October-November 2001.

[36] Keating, M. and Bricaud, P., *Reuse Methodology Manual for System-on-a-Chip Designs*, 2nd Edition, Boston, Kluwer Academic Publisher, 1999.

[37] Kennedy, A. and Syne, D., "Design and Implementation pf Generics for the .Net Common Language Runtime", *Programming Language Design and Implementation*, Snowbird, Utah, pp. 1-12, June 2001.

[38] Khusid, M. and McElrath, L., "StateMate vs. SystemC", 18-849B Project 1, March 26 2001.

[39] Kilgore, R.A., "Multi-language, open-source modeling using the Microsoft .NET architecture", *Winter Simulation Conference*, San Diego, CA, pp. 629-633, December 2002.

[40] Lee, E.A. and Neuendorffer, S., "MoML - A Modeling Markup Language in XML, Version 0.4", Technical Memorandum UCB/ERL M00/12, University of California, Berkeley, 2000.

[41] Liberty J. *"Programming C#: Attributes and Reflection"* ,Oreilley, www.oreillynet.com/pub/a/dotnet/excerpt/prog_csharp_ch18/index.html, 2003.

[42] Lutz, M.H and Laplante, P.A., "C# and the .NET framework: ready for real time?", *IEEE Software*, vol. 20, pp. 74-80, January-February 2003.

[43] Martignano, M, Drago, N., Fummi, F. and Martini, S., "A combined approach to validate the design of embedded network devices", *IEEE International Symposium on Circuits and Systems (ISCAS)*, Scottsdal, AR, pp. 169-172 vol.3, May 2002.

[44] Martin, G., "SystemC and the Future of Design Languages: Opportunities for Users and Research", *Symposium on Integrated Circuits and System Design*, Sao Paulo, Brazil, pp. 61-62, September 2003.

[45] Meijer E., Miller J. Technical *"Overview of the Common Language Runtime"* research.microsoft.com/~emeijer/Papers/CLR.pdf, 2003.

[46] Mono Home Page, www.go-mono.com/, 2003.

[47] Moorby, P., "A Look Back At Verilog", *Electronic Design*, June 10 2002.

[48] Moores Law, www.webopedia.com/TERM/M/Moores_Law.html, 2003.

[49] .NET Framework Home Page, www.microsoft.com/net, 2003.

[50] Newkirk, J. and Vorontsov A.A., "How .NET's custom attributes affect design", *IEEE Software*, vol. 19, pp. 18-20, September-October 2002.

[51] Nicolescu, G. and al., "Validation in a Component-Based Design Flow for Multicore SoCs", *International Symposium on Systems Synthesis*, Kyoto, Japan, pp 162-167, October 2002.

[52] Overview: Hardware Compilation and the Handel-C language web.comlab.ox.ac.uk/oucl/work/christian.peter/overview_handelc.html, 2003.

[53] Paulin, P.G., Pilkington, C. and Bensoudane, E., "StepNP: A System-Level Exploration Platform for Network Processors", *IEEE Design & Test of Computers*, vol. 19, issue 6, pp. 17-26, November-December 2002.

[54] Rich, D.I., "The evolution of SystemVerilog", *IEEE Design & Test of Computers*, vol. 20, issue 4, pp. 82-82 July-August 2003.

[55] Rotor Home Page research.microsoft.com/Collaboration/University/Europe/RFP/Rotor/, 2003.

[56] Sarmenta, L.F.G., Chua, S.J.V., Echevarria, P., Mendoza, J.M.; Santos, R.-R.; Tan, S. and Lozada, R.P., "Bayanihan Computing .NET: grid computing with XML web services", *Cluster Computing and the Grid 2nd IEEE/ACM International Symposium*, Berlin, Germany, pp. 404-405, May 2002.

[57] Shankar, A. "Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API", msdn.microsoft.com/msdnmag/issues/03/09/CoroutinesinNET/default.aspx, 2003

[58] Singer, J., "JVM versus CLR: A Comparative Study", *International Conference on Principles and Practice of Programming in Java*, Kilkenny City, Ireland, pp. 167-169, June 2003.

[59] SpecC Home Page, www.ics.uci.edu/~specc/index.html, 2003.

[60] Synopsys, CoCentric® System Studio, 2001.

[61] SystemC Home Page, www.systemc.org/, 2003.

[62] SystemC in Europe - current usage and future requirements, www.doulos.com/systemc_report/, 2003.

[63] SystemC Version 2 LRM, www.systemc.org, 2003.

[64] SystemC Version 2 User's Guide, www.systemc.org, 2003.

[65] SystemC, Version 2.0, www.systemc.org/, 2003.

[66] T. Grotker, "Modeling software with SystemC 3.0", www-ti.informatik.uni-tuebingen.de/~systemc/ Documents/Presentation-6-OSCI5_groetker.pdf, 2003.

[67] Wikipedia, "Hardware Description Language Definition", en2.wikipedia.org/wiki/Hardware_description_language, 2003.

[68] Yoo, S. and al., "Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer", *Design, Automation and Test in Europe*, Munich, Germany, pp. 150-155, March 2003.

Annex A Fifo Channel Example

```
 1 using System;
 2
 3 // pseudo code; some parts are not shown
 4 // --------------------------------
 5 // INTERFACE : fifo_read_if
 6 // --------------------------------
 7 public interface fifo_read_if{
 8     void read( ref int data ); // blocking read
 9     bool nb_read( ref int data ); // non-blocking read
10     uint num_available();// request #samples available
11 }
12
13 // --------------------------------
14 // INTERFACE : fifo_write_if
15 // --------------------------------
16 public interface fifo_write_if{
17     void write( ref int data ); // blocking write
18     bool nb_write( ref int data ); // non-blocking write
19     uint num_free();// request #spaces free
20 }
21
22 // --------------------------------
23 // PRIMITIVE CHANNEL : fifo
24 // --------------------------------
25 public class fifo: BaseChannel, fifo_read_if, fifo_write_if{
26     int[] mem; // the fifo memory
27     uint front; // size of the fifo
28     uint size; // size of the fifo
29     uint num_readable; // #samples readable
30     uint num_read; // #samples read during this delta cycle
31     uint num_written; // #samples written during this delta cycle
32     Event data_read = new Event();
33     Event data_written = new Event();
34
35     // constructor with size
36     public fifo( uint size ){
37         Verification.Assert( size > 0, "Size must be bigger then 0" );
38         mem = new int[size];
39         this.size = size;
```

```
40      }
41      // blocking read and write access
42      public void read( ref int data ){
43          if( num_available() == 0 )
44              Wait( data_written );
45          nb_read( ref data );
46      }
47      public void write( ref int data ){
48          if( num_free() == 0 )
49              Wait( data_read );
50          nb_write( ref data );
51      }
52      // non-blocking read and write access
53      // return 'true' on success
54      public bool nb_read( ref int data ){
55          if( num_available() == 0 )
56              return false;
57          num_read ++;
58          RequestUpdate();
59          data = mem[front-1];
60          front--;
61          return true;
62      }
63      public bool nb_write( ref int data ){
64          if( num_free() == 0 )
65              return false;
66          num_written ++;
67          RequestUpdate();
68          mem[front]=data;
69          front++;
70          return true;
71      }
72      // request #samples available and #spaces free
73      public uint num_available(){ return (num_readable - num_read); }
74      public uint num_free(){ return (size - num_readable - num_written);}
75
76      public override void Update(){
77          if( num_written > 0 )
78              data_written.Notify(0);
```

```
79          if( num_read > 0 )
80              data_read.Notify(0);
81          num_readable = (uint) mem.Length;
82          num_read = num_written = 0;
83      }
84  }
85
86  // ---------
87  // EXAMPLE
88  // ---------
89  public class writer : BaseModule{
90      public fifo_write_if output;
91
92      [Process]
93      void main_action(){
94          int val = 0;
95          while( true ) {
96              Wait( 10); // wait for 10 ns
97              for( int i = 0; i < 20; i ++ ){
98                  output.write(ref val); // blocking write
99                  val++;
100             }
101         }
102     }
103 }
104
105 public class reader : BaseModule{
106     public fifo_read_if input;
107
108     [Process]
109     void main_action(){
110         int val=0;
111         while( true ) {
112             Wait(10); // wait for 10 ns
113             for( int i = 0; i < 15; i ++ ) {
114                 input.read( ref val); // blocking read
115                 Console.WriteLine(val);
116             }
117         }
```

```
118     }
119 }
120
121 public class model:SystemModel{
122     // declare channel(s)
123     fifo f = new fifo( 10 );
124     // instantiate block(s) and connect to channel(s)
125     writer w = new writer();
126     reader r = new reader();
127
128     public model(ISystemManager manager,String name):base(manager, name){
129         w.output = f;
130         r.input = f;
131     }
132 }
133
134
135 public class App{
136
137     public static void Main(){
138         Simulator sim = new Simulator();
139         model m = new model(sim,"Exemple");
140         sim.Run(1000);
141         Console.ReadLine();
142     }
143 }
```

Annex B Simple Bus Example

```
 1 public enum simple_bus_lock_status
 2 {
 3     SIMPLE_BUS_LOCK_NO = 0,
 4     SIMPLE_BUS_LOCK_SET,
 5     SIMPLE_BUS_LOCK_GRANTED
 6 }
 7
 8 public enum simple_bus_status
 9 {
10     SIMPLE_BUS_OK = 0,
11     SIMPLE_BUS_REQUEST,
12     SIMPLE_BUS_WAIT,
13     SIMPLE_BUS_ERROR
14 }
```

```
1  using System.Collections ;
2
3  public interface simple_bus_arbiter_if
4  {
5      simple_bus_request arbitrate(ArrayList requests);
6  }
7
8  public interface simple_bus_blocking_if
9  {
10     simple_bus_status burst_read(uint unique_priority,  int[] data, uint start_address , bool ulock);
11     simple_bus_status burst_write(uint unique_priority,  int[] data, uint start_address, bool ulock);
12     simple_bus_status burst_read(uint unique_priority,  int[] data, uint start_address);//false
13     simple_bus_status burst_write(uint unique_priority,  int[] data, uint start_address);//false
14 }
15
16
17 public interface simple_bus_direct_if
18 {
19     bool direct_read(ref int data, uint address);
20     bool direct_write(ref int data, uint address);
21 }
22
23
24 public interface simple_bus_non_blocking_if
25 {
26     void read(uint unique_priority, int[] data, uint address, bool ulock);
27     void write(uint unique_priority, int[] data, uint address, bool ulock);
28     void read(uint unique_priority, int[] data, uint address);//false
29     void write(uint unique_priority, int[] data, uint address);//false
30     simple_bus_status get_status(uint unique_priority);
31 }
32
33
34 public interface simple_bus_slave_if:simple_bus_direct_if
35 {
36     simple_bus_status read(ref int data, uint address);
37     simple_bus_status write(ref int data, uint address);
38     uint start_address{get;}
39     uint end_address{get;}
```

```
40 }
41
42
43
44
45
```

```
1  using System;
2  using System.Collections;
3
4  public class simple_bus : BaseChannel,simple_bus_direct_if,simple_bus_non_blocking_if,simple_bus_blocking_if {
5      public Clock clock;
6      public simple_bus_arbiter_if arbiter_port;
7      public ArrayList slave_port = new ArrayList();
8      private bool m_verbose;
9      private ArrayList m_requests = new ArrayList();
10     private simple_bus_request m_current_request=null;
11
12     public  simple_bus(string name_, bool verbose):base(name_){m_verbose=verbose;}
13     public  simple_bus(string name_):base(name_){m_verbose=false;}
14
15     [SimInit]
16     public void end_of_elaboration() {
17         // perform a static check for overlapping memory areas of the slaves
18         bool no_overlap;
19         for (int i = 0; i < slave_port.Count; ++i) {
20         simple_bus_slave_if slave1 =slave_port[i] as simple_bus_slave_if;
21             for (int j = 0; j < i; ++j) {
22                 simple_bus_slave_if slave2 = slave_port[j] as simple_bus_slave_if;
23                 no_overlap = (slave1.end_address < slave2.start_address) || (slave1.start_address > slave2.
       end_address);
24                 if (!no_overlap) {
25                     Console.Out.WriteLine("Error: overlapping address spaces of 2 slaves : ");
26                     Console.Out.WriteLine("slave {0} : {1}X..{2}X",i,slave1.start_address,slave1.end_address);
27                     Console.Out.WriteLine("slave {0} : {1}X..{2}X",j,slave2.start_address,slave2.end_address);
28                     System.Environment.Exit(1);
29                 }
30             }
31         }
32     }
33
34     //-----------------------------------------------------------------------
35     //-- process
36     //-----------------------------------------------------------------------
37     [PMethod]
38     [EventList("negedge","clock")]
```

```
39      public void main_action() {
40          // m_current_request is cleared after the slave is done with a
41          // single data transfer. Burst requests require the arbiter to
42          // select the request again.
43          if (m_current_request == null){
44              m_current_request = get_next_request();
45          }else{
46              // monitor slave wait states
47              if (m_verbose)
48                  Console.Out.WriteLine("{0:g} SLV [{1:d}]", CurrentTime/10.0,m_current_request.address);
49          }
50
51          if (m_current_request != null)
52              handle_request();
53          if (m_current_request==null)
54              clear_locks();
55      }
56
57
58
59      //--------------------------------------------------------------------------
60      //-- direct BUS interface
61      //--------------------------------------------------------------------------
62      public bool direct_read(ref int data, uint address) {
63          if (address%4 != 0 ) {// address not word alligned
64              Console.Out.WriteLine("  BUS ERROR --> address {0}4X not word alligned",address);
65              return false;
66          }
67
68          simple_bus_slave_if slave = get_slave(address);
69
70          if (slave==null)
71              return false;
72
73          return slave.direct_read(ref data, address);
74      }
75
76
77      public bool direct_write(ref int data, uint address) {
```

```
78          if (address%4 != 0 ) {// address not word alligned
79              Console.Out.WriteLine("  BUS ERROR --> address {0}4X not word alligned",address);
80              return false;
81          }
82
83          simple_bus_slave_if slave = get_slave(address);
84
85          if (slave==null)
86              return false;
87
88          return slave.direct_write(ref data, address);
89      }
90
91      //-------------------------------------------------------------------------
92      //-- non-blocking BUS interface
93      //-------------------------------------------------------------------------
94      public void read(uint unique_priority, int[] data, uint address){
95          read(unique_priority,data,address,false);
96      }
97
98      public void read(uint unique_priority, int[] data, uint address, bool ulock) {
99          if (m_verbose)
100             Console.Out.WriteLine("{0:g} {1:s} : read({2:d}) @ {3:x}", CurrentTime/10.0, Name, unique_priority,
        address);
101
102         simple_bus_request request = get_request(unique_priority);
103         // abort when the request is still not finished
104         Verification.Assert((request.status == simple_bus_status.SIMPLE_BUS_OK) ||(request.status ==
        simple_bus_status.SIMPLE_BUS_ERROR),"Error1 in Bus");
105
106         request.do_write = false;
107         request.address = address;
108         request.end_address = address;
109         request.data = data;
110
111         if (ulock)
112             request.ulock = (request.ulock == simple_bus_lock_status.SIMPLE_BUS_LOCK_SET) ?
113                 simple_bus_lock_status.SIMPLE_BUS_LOCK_GRANTED : simple_bus_lock_status.SIMPLE_BUS_LOCK_SET;
114
```

```
115         request.status = simple_bus_status.SIMPLE_BUS_REQUEST;
116     }
117
118     public void write(uint unique_priority, int[] data, uint address){
119         write(unique_priority, data,address,false);
120     }
121
122     public void write(uint unique_priority, int[] data, uint address, bool ulock) {
123         if (m_verbose)
124             Console.Out.WriteLine("{0:g} {1:s} : write({2:d}) @ {3:x}", CurrentTime/10.0, Name, unique_priority,
    address);
125
126         simple_bus_request request = get_request(unique_priority);
127         // abort when the request is still not finished
128         Verification.Assert((request.status == simple_bus_status.SIMPLE_BUS_OK) ||(request.status ==
    simple_bus_status.SIMPLE_BUS_ERROR),"Error2 in Bus");
129
130         request.do_write = true;
131         request.address = address;
132         request.end_address = address;
133         request.data = data;
134
135         if (ulock)
136             request.ulock = (request.ulock == simple_bus_lock_status.SIMPLE_BUS_LOCK_SET) ?
137                 simple_bus_lock_status.SIMPLE_BUS_LOCK_GRANTED : simple_bus_lock_status.SIMPLE_BUS_LOCK_SET;
138
139         request.status = simple_bus_status.SIMPLE_BUS_REQUEST;
140     }
141
142     public simple_bus_status get_status(uint unique_priority) {
143         return get_request(unique_priority).status;
144     }
145
146     //-------------------------------------------------------------------------
147     //-- blocking BUS interface
148     //-------------------------------------------------------------------------
149     public simple_bus_status burst_read(uint unique_priority, int[] data, uint start_address) {
150        return burst_read(unique_priority, data, start_address, false);
151     }
```

```
152
153     public simple_bus_status burst_read(uint unique_priority, int[] data, uint start_address, bool ulock) {
154         if (m_verbose)
155             Console.Out.WriteLine("{0:g} {1:s} : burst-read({2:d}) @ {3:x}", CurrentTime/10.0, Name, unique_priority,↙
      start_address);
156
157         simple_bus_request request = get_request(unique_priority);
158
159         request.do_write = false;
160         request.address = start_address ;
161         request.end_address = start_address + (uint)(data.GetLength(0)-1)*4;
162         request.data = data;
163
164         if (ulock)
165             request.ulock = (request.ulock == simple_bus_lock_status.SIMPLE_BUS_LOCK_SET) ?
166                 simple_bus_lock_status.SIMPLE_BUS_LOCK_GRANTED : simple_bus_lock_status.SIMPLE_BUS_LOCK_SET;
167
168         request.status = simple_bus_status.SIMPLE_BUS_REQUEST;
169
170         Wait(request.transfer_done);
171         Wait(clock.posedge);
172         return request.status;
173     }
174
175     public simple_bus_status burst_write(uint unique_priority, int[] data, uint start_address) {
176         return burst_write(unique_priority, data, start_address, false);
177     }
178
179     public simple_bus_status burst_write(uint unique_priority, int[] data, uint start_address, bool ulock) {
180         if (m_verbose)
181             Console.Out.WriteLine("{0:g} {1:s} : burst-write({2:d}) @ {3:x}", CurrentTime/10.0, Name, unique_priority↙
      , start_address);
182
183         simple_bus_request request = get_request(unique_priority);
184
185         request.do_write = true;
186         request.address = start_address;
187         request.end_address = start_address + (uint)(data.GetLength(0)-1)*4;
188         request.data = data;
```

```
189
190         if (ulock)
191             request.ulock = (request.ulock == simple_bus_lock_status.SIMPLE_BUS_LOCK_SET) ?
192                 simple_bus_lock_status.SIMPLE_BUS_LOCK_GRANTED : simple_bus_lock_status.SIMPLE_BUS_LOCK_SET;
193
194         request.status = simple_bus_status.SIMPLE_BUS_REQUEST;
195
196         Wait(request.transfer_done);
197         Wait(clock.posedge);
198         return request.status;
199     }
200
201     //-------------------------------------------------------------------------
202     //-- BUS methods:
203     //
204     //     handle_request()    : performs atomic bus-to-slave request
205     //     get_request()       : BUS-interface: gets the request form of given
206     //                           priority
207     //     get_next_request()  : returns a valid request out of the list of
208     //                           pending requests
209     //     clear_locks()       : downgrade the lock status of the requests once
210     //                           the transfer is done
211     //-------------------------------------------------------------------------
212     public void handle_request() {
213         if (m_verbose)
214             Console.Out.WriteLine("{0:g} {1:s} Handle Slave({2:d})",CurrentTime/10.0, Name, m_current_request.
    priority);
215
216         m_current_request.status = simple_bus_status.SIMPLE_BUS_WAIT;
217         simple_bus_slave_if slave = get_slave(m_current_request.address);
218
219         if ((m_current_request.address)%4 != 0 ) {
220             Console.Out.WriteLine("  BUS ERROR --> address {0}4X not word alligned",m_current_request.address);
221             m_current_request.status = simple_bus_status.SIMPLE_BUS_ERROR;
222             m_current_request = null;
223             return;
224         }
225
226         if (slave==null) {
```

```
227             Console.Out.WriteLine("  BUS ERROR --> no slave for address {0}4X",m_current_request.address);
228             m_current_request.status = simple_bus_status.SIMPLE_BUS_ERROR;
229             m_current_request = null;
230             return;
231         }
232
233         simple_bus_status slave_status = simple_bus_status.SIMPLE_BUS_OK;
234
235         if (m_current_request.do_write)
236             slave_status = slave.write(ref m_current_request.data[m_current_request.index],m_current_request.address)↙
    ;
237         else
238             slave_status = slave.read(ref m_current_request.data[m_current_request.index],m_current_request.address);
239
240         if (m_verbose)
241             Console.Out.WriteLine("  --> status=({0})", slave_status);
242
243         switch(slave_status) {
244             case simple_bus_status.SIMPLE_BUS_ERROR:
245                 m_current_request.status = simple_bus_status.SIMPLE_BUS_ERROR;
246                 m_current_request.transfer_done.Notify();
247                 m_current_request = null;
248                 break;
249             case simple_bus_status.SIMPLE_BUS_OK:
250                 m_current_request.address+=4; //next word (byte addressing)
251                 m_current_request.index++;
252                 if (m_current_request.address > m_current_request.end_address) {
253                     // burst-transfer (or single transfer) completed
254                     m_current_request.status = simple_bus_status.SIMPLE_BUS_OK;
255                     m_current_request.transfer_done.Notify(0);
256                     m_current_request = null;
257                 }
258                 else {
259                     // more data to transfer, but the (atomic) slave transfer is done
260                     m_current_request = null;
261                 }
262                 break;
263             case simple_bus_status.SIMPLE_BUS_WAIT:
264                 // the slave is still processing: no clearance of the current request
```

```
265                 break;
266             default:
267                 break;
268         }
269     }
270
271     public simple_bus_slave_if get_slave(uint address) {
272         foreach(simple_bus_slave_if slave in slave_port) {
273             if ((slave.start_address <= address) &&(address <= slave.end_address))
274                 return slave;
275         }
276         return null;
277     }
278
279     public simple_bus_request get_request(uint priority) {
280         foreach(simple_bus_request req in m_requests)
281             if (req.priority == priority){
282                 req.index=0;
283                 return req;
284             }
285         simple_bus_request request = new simple_bus_request((ISystemManager)manager);
286         request.priority = priority;
287         m_requests.Add(request);
288         return request;
289     }
290
291     public simple_bus_request get_next_request() {
292         // the slave is done with its action, m_current_request is
293         // empty, so go over the bag of request-forms and compose
294         // a set of likely requests. Pass it to the arbiter for the
295         // final selection
296         ArrayList Q = new ArrayList();
297         foreach(simple_bus_request req in m_requests) {
298             if ((req.status == simple_bus_status.SIMPLE_BUS_REQUEST) ||(req.status == simple_bus_status.
    SIMPLE_BUS_WAIT)) {
299                 if (m_verbose)
300                     Console.Out.WriteLine("{0:g} {1} : request ({2:d}) [{3}]",CurrentTime/10.0, Name, req.priority,
    req.status);
301                 Q.Add(req);
```

```
302             }
303         }
304         if (Q.Count > 0)
305             return arbiter_port.arbitrate(Q);
306         return null;
307     }
308
309     public void clear_locks() {
310         foreach(simple_bus_request req in m_requests) {
311             if (req.ulock == simple_bus_lock_status.SIMPLE_BUS_LOCK_GRANTED)
312                 req.ulock = simple_bus_lock_status.SIMPLE_BUS_LOCK_SET;
313             else
314                 req.ulock = simple_bus_lock_status.SIMPLE_BUS_LOCK_NO;
315         }
316     }
317 }
318
319
320
```

```
1  using System;
2  using System.Collections;
3
4  public class simple_bus_arbiter: BaseModule, simple_bus_arbiter_if
5  {
6      private bool m_verbose;
7
8      public simple_bus_arbiter(string name_): base(name_){m_verbose = false;}
9      public simple_bus_arbiter(string name_, bool verbose): base(name_){m_verbose = verbose;}
10
11
12     public simple_bus_request arbitrate(ArrayList requests)
13     {
14         if (m_verbose)
15         {
16             // shows the list of pending requests
17             Console.Out.Write("{0:g} {1} :", CurrentTime/10.0, Name);
18              char[] lock_chars = { '-', '=', '+' };
19             foreach(simple_bus_request request in requests)
20             {
21                 Console.Out.Write("\n    R[{0:d}]({1}{2}@{3:x})",request.priority,lock_chars[(int)request.ulock],
       request.status,request.address);
22             }
23         }
24         simple_bus_request best_request = requests[0] as simple_bus_request;
25
26         foreach(simple_bus_request request in requests)
27             if ((request.status == simple_bus_status.SIMPLE_BUS_WAIT)&&(request.ulock == simple_bus_lock_status.
       SIMPLE_BUS_LOCK_SET))
28             {
29                 if (m_verbose)
30                     Console.Out.WriteLine(" -> R[{0:d}] (rule 1)", request.priority);
31                 return request;
32             }
33         foreach(simple_bus_request request in requests)
34             if (request.ulock == simple_bus_lock_status.SIMPLE_BUS_LOCK_GRANTED)
35             {
36                 if (m_verbose)
37                     Console.Out.WriteLine(" -> R[{0:d}] (rule 2)", request.priority);
```

```
38                  return request;
39              }
40          foreach(simple_bus_request request in requests)
41          {
42              //sc_assert(requests[i]->priority != best_request->priority);
43              if (request.priority < best_request.priority)
44                  best_request = request;
45          }
46
47          if (best_request.ulock != simple_bus_lock_status.SIMPLE_BUS_LOCK_NO)
48              best_request.ulock = simple_bus_lock_status.SIMPLE_BUS_LOCK_GRANTED;
49
50          if (m_verbose)
51              Console.Out.WriteLine(" -> R[{0:d}] (rule 3)", best_request.priority);
52
53          return best_request;
54      }
55  }
56
57
58
59
```

```
1 using System;
2
3 public class simple_bus_fast_mem : BaseModule, simple_bus_slave_if
4 {
5     public Clock clock;
6     private int[] MEM;
7     private uint m_start_address;
8     private uint m_end_address;
9
10    public simple_bus_fast_mem(string name_, uint start_address, uint end_address): base(name_){
11        m_start_address = start_address;
12        m_end_address = end_address;
13
14        Verification.Assert(m_start_address <= m_end_address,"FastMem error1");
15        Verification.Assert((m_end_address-m_start_address+1)%4 == 0,"FastMem error2");
16
17        uint size = (m_end_address-m_start_address+1)/4;
18        MEM = new int [size];
19        for (uint i = 0; i < size; ++i)
20            MEM[i] = 0;
21    }
22
23    public uint start_address {
24        get{return m_start_address;}
25    }
26
27    public uint end_address{
28        get{return m_end_address;}
29    }
30
31
32 //    [PMethod]
33 //    [EventList("posedge","clock")]
34 //    public void wait_loop() {
35 //        for(int i=0;i<32;i++){
36 //            Console.WriteLine("{0:x}",MEM[i]);
37 //        }
38 //        Console.WriteLine("****************");
39 //    }
```

```
40
41      public bool direct_read(ref int data, uint address)
42      {
43          return (read(ref data, address) == simple_bus_status.SIMPLE_BUS_OK);
44      }
45
46      public bool direct_write(ref int data, uint address)
47      {
48          return (write(ref data, address) == simple_bus_status.SIMPLE_BUS_OK);
49      }
50
51      public simple_bus_status read(ref int data, uint address)
52      {
53              data=MEM[(address - m_start_address)/4];
54          return simple_bus_status.SIMPLE_BUS_OK;
55      }
56
57      public simple_bus_status write(ref int data, uint address)
58      {
59              MEM[(address - m_start_address)/4] = data;
60          return simple_bus_status.SIMPLE_BUS_OK;
61      }
62  }
63
64
65
66
67
```

```
1  public class simple_bus_slow_mem : BaseModule, simple_bus_slave_if {
2      public Clock clock;
3      private int[] MEM;
4      private uint m_start_address;
5      private uint m_end_address;
6      private uint m_nr_wait_states;
7      private int m_wait_count;
8
9      public simple_bus_slow_mem(string name_, uint start_address, uint end_address, uint nr_wait_states): base(name_) {
10         m_start_address = start_address;
11         m_end_address = end_address;
12
13         Verification.Assert(m_start_address <= m_end_address,"SlowMem error1");
14         Verification.Assert((m_end_address-m_start_address+1)%4 == 0,"SlowMem error2");
15         m_nr_wait_states=nr_wait_states;
16         m_wait_count=-1;
17         uint size = (m_end_address-m_start_address+1)/4;
18         MEM = new int [size];
19         for (uint i = 0; i < size; ++i)
20             MEM[i] = 0;
21     }
22
23     public uint start_address {
24         get{return m_start_address;}
25     }
26
27     public uint end_address{
28         get{return m_end_address;}
29     }
30
31     [PMethod]
32     [EventList("posedge","clock")]
33     public void wait_loop() {
34         if (m_wait_count >= 0) m_wait_count--;
35     }
36
37     public bool direct_read(ref int data, uint address) {
38         data  = MEM[(address - m_start_address)/4];
39         return true;
```

```
40      }
41
42      public bool direct_write(ref int data, uint address) {
43          MEM[(address - m_start_address)/4]= data;
44          return true;
45      }
46
47      public simple_bus_status read(ref int data, uint address) {
48          if (m_wait_count < 0) {
49              m_wait_count = (int)m_nr_wait_states;
50              return simple_bus_status.SIMPLE_BUS_WAIT;
51          }
52          if (m_wait_count == 0) {
53              data = MEM[(address - m_start_address)/4];
54              return simple_bus_status.SIMPLE_BUS_OK;
55          }
56          return simple_bus_status.SIMPLE_BUS_WAIT;
57      }
58
59      public simple_bus_status write(ref int data, uint address) {
60          if (m_wait_count < 0) {
61              m_wait_count = (int)m_nr_wait_states;
62              return simple_bus_status.SIMPLE_BUS_WAIT;
63          }
64          if (m_wait_count == 0) {
65              MEM[(address - m_start_address)/4] = data;
66              return simple_bus_status.SIMPLE_BUS_OK;
67          }
68          return simple_bus_status.SIMPLE_BUS_WAIT;
69      }
70  }
71
72
```

```
1  using System;
2
3  public class simple_bus_master_blocking: BaseModule
4  {
5
6      public Clock clock;
7      public simple_bus_blocking_if bus_port;
8      private uint m_unique_priority;
9      private uint m_address;
10     private bool m_lock;
11     private int m_timeout;
12     private const uint mylength = 0x10;
13
14     public simple_bus_master_blocking(string name_, uint unique_priority, uint address, bool ulock, int timeout):base ↙
       (name_)
15     {
16         m_unique_priority=unique_priority;
17         m_address=address;
18         m_lock=ulock;
19         m_timeout=timeout;
20     }
21
22     [Process]
23     [EventList("posedge","clock")]
24     public void main_action()
25     {
26         int[] mydata = new int[mylength];
27         int i;
28         simple_bus_status status;
29         while (true){
30             //Wait();
31             status = bus_port.burst_read(m_unique_priority, mydata, m_address, m_lock);
32             if (status == simple_bus_status.SIMPLE_BUS_ERROR)
33                 Console.Out.WriteLine("{0:g} {1:s} : blocking-read failed at address {2:x}",CurrentTime/10.0, Name,   ↙
       m_address);
34
35             for (i = 0; i < mylength; ++i){
36                 mydata[i] += i;
37                 Wait();
```

```
38              }
39
40              status = bus_port.burst_write(m_unique_priority, mydata, m_address, m_lock);
41              if (status == simple_bus_status.SIMPLE_BUS_ERROR)
42                  Console.Out.WriteLine("{0:g} {1:s} : blocking-write failed at address {2:x}",CurrentTime/10.0, Name,
    m_address);
43
44              Wait(m_timeout);
45              //Wait();
46          }
47      }
48  }
49
50
51
52
53
```

```
1  using System;
2
3  public class simple_bus_master_non_blocking : BaseModule
4  {
5      public Clock clock;
6      public simple_bus_non_blocking_if bus_port;
7      private uint m_unique_priority;
8      private uint m_start_address;
9      private bool m_lock;
10     private int m_timeout;
11
12     public simple_bus_master_non_blocking(string _name, uint unique_priority, uint start_address, bool ulock, int
       timeout): base(_name)
13     {
14         m_unique_priority=unique_priority;
15         m_start_address=start_address;
16         m_lock=ulock;
17         m_timeout=timeout;
18     }
19
20     [Process]
21     [EventList("posedge","clock")]
22     public void main_action()
23     {
24         int[] mydata= {0};
25         int cnt = 0;
26         uint addr = m_start_address;
27         //Wait(); // ... for the next rising clock edge
28         while (true) {
29             bus_port.read(m_unique_priority, mydata, addr, m_lock);
30             while ((bus_port.get_status(m_unique_priority) != simple_bus_status.SIMPLE_BUS_OK) &&(bus_port.get_status
       (m_unique_priority) != simple_bus_status.SIMPLE_BUS_ERROR))
31                 Wait();
32
33             if (bus_port.get_status(m_unique_priority) == simple_bus_status.SIMPLE_BUS_ERROR)
34                 Console.Out.WriteLine("{0:g} {1:s} : ERROR cannot read from {2:x}",CurrentTime/10.0, Name, addr);
35
36             mydata[0] += cnt;
37             cnt++;
```

```
38
39              bus_port.write(m_unique_priority, mydata, addr, m_lock);
40              while ((bus_port.get_status(m_unique_priority) != simple_bus_status.SIMPLE_BUS_OK)&&(bus_port.get_status
    (m_unique_priority) != simple_bus_status.SIMPLE_BUS_ERROR))
41                  Wait();
42
43              if (bus_port.get_status(m_unique_priority) == simple_bus_status.SIMPLE_BUS_ERROR)
44                  Console.Out.WriteLine("{0:g} {1:s} : ERROR cannot write to {2:x}",CurrentTime/10.0, Name, addr);
45
46              Wait(m_timeout);
47              //Wait(); // ... for the next rising clock edge
48              addr+=4; // next word (byte addressing)
49              if (addr > (m_start_address+0x80)) {
50                  addr = m_start_address; cnt = 0;
51              }
52          }
53      }
54  }
55
56
57
58
59
60
```

```
using System;

public class simple_bus_master_direct : BaseModule
{
    public Clock clock;
    public simple_bus_direct_if bus_port;
    private uint m_address;
    private int m_timeout;
    private bool m_verbose;

    public simple_bus_master_direct(string name_, uint address, int timeout, bool verbose): base(name_)
    {
        m_address=address;
        m_timeout=timeout;
        m_verbose=verbose;
    }

    public simple_bus_master_direct(string name_, uint address, int timeout): base(name_)
    {
        m_address=address;
        m_timeout=timeout;
        m_verbose=false;
    }

    [Process]
    [EventList("posedge","clock")]
    public void main_action()
    {
        int[] mydata = new int[4];
        while (true)
        {
            bus_port.direct_read(ref mydata[0], m_address);
            bus_port.direct_read(ref mydata[1], m_address+4);
            bus_port.direct_read(ref mydata[2], m_address+8);
            bus_port.direct_read(ref mydata[3], m_address+12);

            if (m_verbose)
                Console.WriteLine("{0:g} {1:s} : mem[{2:x}:{3:x}] = ({4:x}, {5:x}, {6:x}, {7:x})",CurrentTime/10.0,
    Name, m_address, m_address+15, mydata[0], mydata[1], mydata[2], mydata[3]);
```

```
39              Wait(m_timeout);
40          }
41      }
42  }
43
44
45
46
47
```

Annex C My First System Example

```
1 //ThirdPartyToolExample.cs
2 //By
3 //James Lapalme 2003
4
5 using System;
6 using System.Collections;
7 using System.Reflection;
8
9 ///<summary>Synchronous Interger Generator</summary>
10 ///<remarks>
11 ///Instances of this component, when driving by a clock,
12 ///generate an integer value on each positive
13 ///edge of the clock.
14 ///
15 ///The value is incremented by 1 after each
16 ///positive edge
17 ///</remarks>
18
19 public class ModuleA : BaseModule{
20
21     /// <summary>Clock input port</summary>
22     public Clock clk;
23     /// <summary>Integer ouput port</summary>
24     public outInt porta;
25
26     /// <summary>Component constructor </summary>
27     /// <param name="name">Name given to the constructed instance</param>
28     public ModuleA(string name): base(name){}
29
30     ///<summary>Integer generating process method</summary>
31     ///<remarks>This process method is sensitive to the posedge event of the clk port</remarks>
32     [Process]
33     [EventList("posedge","clk")]
34     public void Gen() {
35         while(true){
36             for(int i=0;0<100;i++){
37                 porta.Value=i;
38                 Wait();
39             }
40         }
41     }
```

```
42 }
43
44 ///<summary>Transactional Level Testbench</summary>
45 ///<remarks>
46 ///Instances of this component are synchonized
47 ///by the means of transactional level event notification
48 ///
49 ///When notified of a new value on its input port,
50 ///the component reads the value and outputs it to the screen
51 ///</remarks>
52 public class ModuleB : BaseModule {
53
54     /// <summary>Integer input port</summary>
55     public inInt porta;
56
57     /// <summary>Synchronization event</summary>
58     public Event syn_event;
59
60     /// <summary>Component constructor </summary>
61     /// <param name="name">Name given to the constructed instance</param>
62     public ModuleB(string name): base(name){}
63
64      ///<summary>outputing process method</summary>
65     [Process]
66     public void Run() {
67         while(true){
68             Wait(syn_event);
69             Console.WriteLine(porta.Value);
70         }
71     }
72 }
73
74
75 ///<summary>Component Under Test</summary>
76 ///<remarks>
77 ///Instances of this component, when driving by a clock,
78 ///generate a even integer value on each positive
79 ///edge of the clock.
80 ///
81 //////The value is incremented by 2 after each
82 ///positive edge
```

```
 83 ///</remarks>
 84 public class MyModule : BaseModule {
 85     /// <summary>Clock input port</summary>
 86     public Clock clk;
 87     /// <summary>Integer output port</summary>
 88     public outInt porta;
 89     /// <summary>Synchronization event</summary>
 90     public Event syn_event = new Event();
 91     /// <summary>Sub-component instance</summary>
 92     ModuleA gen1 = new ModuleA("gen1");
 93     /// <summary>Sub-component instance</summary>
 94     ModuleA gen2 = new ModuleA("gen2");
 95     /// <summary>Inner signal instance</summary>
 96     IntSignal sig1 = new IntSignal();
 97     /// <summary>Inner signal instance</summary>
 98     IntSignal sig2 = new IntSignal();
 99
100     /// <summary>Component constructor </summary>
101     /// <param name="name">Name given to the constructed instance</param>
102     public MyModule(string name): base(name){
103         gen1.porta = sig1;
104         gen2.porta = sig2;
105     }
106
107     ///<summary>Integer generating process method</summary>
108     ///<remarks>This process method is sensitive to the
109     ///sensitive event of the two inne signal sig1 and sig2</remarks>
110     [PMethod]
111     [EventList("sensitive","sig1","sig2")]
112     public void Add() {
113         porta.Value = sig1.Value + sig2.Value;
114         syn_event.Notify(0);
115     }
116
117     ///<summary>BindingPhase method</summary>
118     ///<remarks>Binding of the clock signal to the sub-components is done here</remarks>
119     public override void BindingPhase(){
120         gen1.clk = clk;
121         gen2.clk = clk;
122     }
123 }
```

```
124
125 ///<summary>Synchronous Interger Generator</summary>
126 ///<remarks>
127 ///Instances of this component, when driving by a clock,
128 ///generate an integer value on each positive
129 ///edge of the clock.
130 ///
131 ///The value is incremented by 1 after each
132 ///positive edge
133 ///</remarks>
134 public class MyFirstSystem:SystemModel{
135    /// <summary>Generator component</summary>
136     MyModule top = new MyModule("generator");
137     /// <summary>Testbench component</summary>
138     ModuleB testbench = new ModuleB("testbench");
139     /// <summary>SMain system clock</summary>
140     Clock clk = new Clock("clock1",4);
141     /// <summary>Inner signal</summary>
142     IntSignal sig3 = new IntSignal();
143
144     /// <summary>System model constructor</summary>
145     public MyFirstSystem (ISystemManager manager): base(manager){
146         top.clk = clk;
147         top.porta = sig3;
148         testbench.porta = sig3;
149         testbench.syn_event = top.syn_event;
150     }
151 }
152
153
154 ///<summary>Verification Tool</summary></summary>
155 ///<remarks>
156 ///Instances of this component discover dynamically
157 ///the signal and ports of a system model.
158 ///</remarks>
159 public class SignalPrinter{
160     ArrayList signals = new ArrayList();
161     SystemModel systemModel;
162     string systemName;
163
164     ///<summary>Verification Tool</summary>
```

```
165     ///<param name="model">This is the bound system model instance</param>
166     ///<param name="name">This is the name of the system model instance</param>
167     public SignalPrinter(SystemModel model, string name){
168         systemModel = model;
169         systemName=name;
170     }
171
172     /// <summary>Class method used has a callback to initialize the tool</summary>
173     /// <remarks>Initialization of the tools cause the discovery of the system model</remarks>
174     public void Initialize(){
175         DiscoverSignals(systemModel,systemName);
176     }
177     /// <summary>Class method used for signal and port discovery</summary>
178     private void DiscoverSignals(IModelElementContainer element,string parentname) {
179         Type type = element.GetType();
180         ArrayList hierarchialElements = new ArrayList();
181         Object[] couple;
182         foreach(FieldInfo fi in type.GetFields(BindingFlags.Instance|
183             BindingFlags.Public  |
184             BindingFlags.NonPublic)){
185             Object obj = fi.GetValue(element);
186             if(obj is BaseModule){
187                 couple = new Object[2];
188                 couple[0] = parentname +  "." + fi.Name;
189                 couple[1] = obj;
190                 hierarchialElements.Add(couple);
191             }else if(obj is BaseSignal){
192                 couple = new Object[2];
193                 couple[0] = parentname +  "." + fi.Name;
194                 couple[1] = obj;
195                 signals.Add(couple);
196             }
197         }
198         foreach(Object[] pair in hierarchialElements)
199             DiscoverSignals((IModelElementContainer)pair[1],(string)pair[0]);
200     }
201     /// <summary>Class method used as a callback to print the current
202     /// values of the discovered system model signals and ports</summary>
203     public void PrintSignals(){
204         foreach(Object[] pair in signals){
205             Object currentValue = pair[1].GetType().GetProperty("Value").GetValue(pair[1],null);
```

```
206             Object nextValue = pair[1].GetType().GetProperty("NextValue").GetValue(pair[1],null);
207             Console.WriteLine("----------------------------");
208             Console.WriteLine("Signal/Port name: {0};",pair[0]);
209             Console.WriteLine("Current value: {0};",currentValue);
210             Console.WriteLine("----------------------------");
211         }
212     }
213 }
214 ///<summary>Main entry point application</summary>
215 ///<remarks>
216 ///The system model, verification tool and simulator are instantiated and bound togheter.
217 ///The simulator is then started for 20 units of time.
218 ///</remarks>
219 public class App {
220     /// <summary>
221     /// Application entry point
222     /// </summary>
223     static void Main(string[] args) {
224         Simulator sim = new Simulator();
225         MyFirstSystem sys = new MyFirstSystem(sim);
226         SignalPrinter sp = new SignalPrinter(sys,"MyFirstSystem");
227         sim.simInit+= new RunnableMethod(sp.Initialize);
228         sim.cycleInit+= new RunnableMethod(sp.PrintSignals);
229         sim.Run(20);
230     }
231 }
232
```