VTT PUBLICATIONS 355

A VHDL simulator in a co-verification environment

Klaus Melakari
VTT Electronics

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 1998

ISBN 951–38–5245–8 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–5246–6 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Elektroniikan piirit ja järjestelmät, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Elektroniska kretsar och system, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Electronic Circuits and Systems,
Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Leena Ukskoski

LIBELLA PAINOPALVELU OY, ESPOO 1998

Melakari, Klaus. A VHDL simulator in a co-verification environment. Espoo 1998, Technical Research Centre of Finland, VTT Publications 355. 91 p. + app. 6 p.

Keywords VHDL, simulators, co-simulation, embedded systems

ABSTRACT

In this work, methods and tools are developed for the integration of a VHDL simulation environment into the Modelling and Simulation Environment (MSE) of the ESPRIT project EP20576 (OMI/TOOLS). The MSE is a co-verification environment for the verification of mixed hardware/software systems that eventually can be implemented as system-chips. System-chips are integrated circuits consisting of processors, memories, software components and application specific hardware parts. The VHDL simulation environment constitutes that part of the MSE which is intended for the simulation of the hardware part in VHDL of the mixed hardware/software system. The software part is modelled in C and it is simulated by a ClearSim software simulator. In addition to simulators, the MSE contains the Graphical Model Builder, the Model Database, the InterOperation Engine and the Graphical Animator to provide means for reusability and efficient handling of the system-level issues.

The contribution of this work to the MSE development is the specification, design, implementation and testing of the methods and programs that enable integration of the VHDL environment into the MSE. The integration requires an interface to the modelling and simulation backplane.

The functionality of the integrated VHDL simulator was tested by simulating two hardware/software models. The results were then compared with current solutions and industrial expectations. The key result was that the MSE integrated VHDL simulator can provide the simulation of the hardware portion in the co-simulation of mixed hardware/software systems. The estimations of achievable performance assessed that the MSE, as a whole, enables much faster co-simulation than any currently existing commercial solution.

PREFACE

This a M.Sc. thesis based on the research project carried out at VTT Electronics in the OMI/TOOLS project during the years 1996 - 98. The OMI/TOOLS project was part of the European Community ESPRIT IV programme and it was carried out with co-operating partners around Europe.

I wish to thank the supervisors of this thesis Professor Juha Kostamovaara and PhD. Hannu Heusala.

I would also like to thank Juha-Pekka Soininen, Marko Heikkinen and Kari Tiensyrjä for their comments and criticism. Especially I am indebted to Janne Saarikettu, who helped to formulate the contents of this thesis to more readable form.

I am also grateful to OMI/TOOLS partners for smooth co-operation during the integration and development of the MSE. Especially I would like to thank Jochen Bruns and Werner van Almsick for their instant response to my questions and ideas.

CONTENTS

LIST OF ACRONYMS

API	Application Programming Interface, a function library containing interfacing functions for application.
ASIC	Application-Specific Integrated Circuit, an integrated circuit, which is specially designed for one application.
C	Commonly used programming language.
ClearSim	Execution driven, instruction level software simulator.
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ESPRIT	European Strategic Programme for Research and Development in Information Technology
Ethernet	IEEE 802.3 Communication network
FLI	Foreign Language Interface, C-function library, enabling the execution of C-processes in the QuickHDL simulator.
FPGA	Field Programmable Gate-Array, programmable logic device.
GA	Graphical Animator
GMB	Graphical Model Builder
GNU	Gnu's Not UNIX, UNIX compatible software system, which is available for free.
HDL	Hardware Description Language

HW	HardWare
ICLib	IOE/IEI Communication Library, API for ICTP.
ICTP	InterOperation engine Communication Transmission Protocol, a protocol that is used in the tool to tool communication during the simulation.
IEI	InterOperation Engine Interface, the MSE integrated tools' interface to the IOE.
IOE	InterOperation Engine, the simulation backplane of the MSE.
ISS	Instruction Set Simulator, simulator that enables the execution of target processors code in the host workstation.
makefile	File containing the dependency rules for UNIX make program
MDB	Model DataBase, a database for storing the modelling data of the MSE.
MIF	Model InterFace, interface to common modelling of the MSE
MSE	Modelling and Simulation Environment, environment that is developed during the OMI/TOOLS project.
OMI	Open Microprocessor systems Initiative, a focused cluster in the ESPRIT IV.
OMI/TOOLS	Project for developing tools for validation and observation of embedded systems
processor core	microprocessor which can be inserted to ASIC as a macrocell
Prosa	commercial graphical editor

QuickHDL	VHDL simulator
RTL	Register Transfer Level
SCG	Simulation Configuration Generator
SQL	Structured Query Language
SW	SoftWare
TCP/IP	widely used communication protocol.
Verilog	Hardware description language
VHDL	Hardware description language

1. INTRODUCTION

The majority of the systems designed today include small computer devices, embedded systems, as a functional part. Still the trend of market is an ever increasing complexity and amount of embedded systems. This trend can be seen as ranging from aeroplanes to exercise bicycles, industry plants, ordinary kitchens, etc.

The increasing complexity of electronic systems has forced Electronic Design Automation (EDA) vendors to develop tools and methods to ease the hardware design. Similarly the software development, especially in the workstation and personal computer field has led to sophisticated development environments. However, software and hardware development tools are developed separately to meet the needs of the markets, but the link between these two has been largely ignored. Still it is well known that the time spent in integration of hardware and software can be a significant part of the total development time and costs. While it has been possible to execute small portions of software on the simulated hardware, the speed and usability has been totally inadequate for typical complex software used in embedded systems.

The trend towards deeper integration has also worsened the situation, since integration has increased the amount of custom hardware, such as Digital Signal Processing (DSP)-hardware macro cells and core-processors, to be integrated into the same chip. It is no longer possible to optimise the performance and to verify the design by just emulating the target processor; instead, the simulation of the whole system is required. Simultaneously with the increasing complexity and integration of the systems, the product life cycle, and similarly the development time, has become shorter, reducing the time allowed for system integration and final testing of the product.

During recent years, the increase in the workstation performance and the development in simulator technology have made it possible to apply concurrent simulation of mixed hardware and software systems. Co-simulation gives the designer a possibility to verify the behaviour of the system during the development, thus reducing the time required in the final integration phase.

This thesis describes the embedding of a VHDL simulator into a co-simulation environment, the Modelling and Simulation Environment (MSE) of the OMI/TOOLS project. Furthermore, this thesis gives the reader not only an idea of how co-simulation is achieved in this particular environment, but also an idea of why co-simulation is needed, what are its problems and how they can be solved.

The OMI/TOOLS project is a part of the European Union's ESPRIT IV programme. ESPRIT IV is the largest research programme in the European Union and it is aimed at improving the technologies important to European industry.

The OMI/TOOLS project aims at providing tools and methods for combined hardware/software systems development, especially for real-time embedded control systems. The method used is to develop tools that enable existing development environments to be integrated to each other.

The rest of this thesis is organised as follows: In chapter 2 the problems and methods related to verification of the integrated systems are described briefly, in order to give reader the background needed to understand the objective of this work. Co-simulation as a solution to verification is described in more detail in chapter 3. Also the current status of the co-simulation products provided by Electronic Design Automation (EDA) vendors today is introduced briefly. The OMI/TOOLS project, of which this work forms a part, is introduced in chapter 4. This chapter gives the reader the idea behind the methods selected for embedding the VHDL simulator in the MSE. The writer's contribution to the OMI/TOOLS project, embedding of a VHDL simulator, is described in chapter 5. The results of the MSE simulation and modelling are evaluated in chapter 6.

2. BACKGROUND TO SYSTEMS-ON-SILICON

The price of the silicon chip is decreasing, as the integration density is growing, allowing more complex systems to be integrated in to one chip. The capacity of the silicon chip is increasing by about 50% a year. This is achieved by increasing the size of the chip and simultaneously increasing the level of integration.

Traditionally, the whole silicon capacity is used to build standard components, such as microprocessors and memories. The partitioning of the system has been bound to existing hardware target architecture. This could be called FAT-WARE due to the excessive usage of hardware resources. Typically FAT-WARE embedded systems are workstations, personal computers and industrial automation systems.

When the silicon capacity of a single silicon chip is divided between several components of the system, such as core processors, memories, data paths and interfacing logic, a system-on-silicon is achieved. The main driving force towards systems-on-silicon is the high cost of manufacturing circuit boards and their high tendency to mechanical errors, such as bad soldering. Another reason for the transfer to complex systems-on-silicon is the limited capabilities of printed circuit boards to connect different parts of the system to each other, since the amount of signals and frequency used when communicating between two separate chips is lower and takes much more power than two parts of the system inside one chip [1].

The system-on-silicon allows complex and otherwise expensive embedded systems to be used even in ordinary consumer electronic devices, since the cost of the single chip is lower than the same system implemented with discrete processors and memories. However, the design of a system-on-silicon is more demanding than that of an ordinary system, since although the price of chip in large quantities is low, the cost of the prototype is high. This means that a system-on-chip requires even more careful design than ordinary systems.

Another problem involved in system-on-silicon design is the SW/HW partitioning. The performance of the whole system cannot be directly derived

from the performance of the core processor used, but rather the whole architecture of the chip. The performance and similarly the cost effectiveness must be achieved by careful co-design of the system as a whole. The system should not be considered as separate processors, separate random logic and software. This is mostly due the fact that the system has internal shared resources, such as memories, data busses, I/O ports and different kinds of device drivers, and the correct and efficient sharing of these resources is the key to achieving the desired performance. System-on-silicon type designs are called CO-WARE. Currently the CO-WARE is an increasing trend, and typical users are the hand-held products, such as mobile phones, pen computers and electronic calendars.

When the performance needs of the product are kept constant, ever increasing integration allows the systems to be smaller. Small size makes it possible to use embedded systems in products, without changing their size or appearance. This trend is called THIN-WARE, and its typical implementations are electronic watches and smart cards [2].

2.1 CO-DESIGN

Co-design is a design concept which aims at concurrent development of hardware and software to obtain a complete system design. The idea behind co-design is to design the system as a whole, in order to achieve desired performance and functionality rather than developing individual parts of the system separately and then later putting a great deal of effort into integrating them.

The reasons for using co-design are that the systems are gradually getting so complex that the integration of the different parts together takes too large a portion of the total system development time. Additionally, integration of the almost ready developed parts may reveal errors which cause the designer to loop back to developing and debugging individual hardware/software partitions. These time consuming iterations are not in line with today’s increasing demand for shorter development times.

The development of the embedded systems towards the system-on-silicon structures has also added interest in co-design since these systems allow system optimisation to a much larger extend than traditional designs made from of the shelf components. Although the core-processors used in the system-on-silicon chips are typically slower than their single chip counterparts, the overall performance can be significantly better and the price in the final product smaller if enough system-level optimisation is used [1, 3, 4].

The design process in co-design can be divided into four phases [3]:

1. Functional system specification, which has no implementation specific details.

2. Partitioning of the system into parts to be developed in different technologies. The system-level description is then verified against the functional system specification.

3. The register-transfer (RT) level description of the system, which consist of optimised C code for the software parts and RT-level netlists for the hardware parts of the system. Parts of the system are then verified against the system-level description of phase 2.

Manufacturing data for parts of the system and the system as a whole. The results are then verified against the RT-level description

2.2 CO-VALIDATION

During the design process, the design requires several validation and verification phases, in order to ensure that the final product still fulfills the specifications. The term *validation* means that the current phase product is tested against the specification of the system. Typically, this is done by simulating the phase product and comparing the results to specified values. The term *verification* means that the current phase product is compared to the preceding phase product in order to ensure that the consistency between the phase products is achieved. If the verification is done after each phase product is finished, the cost of revealing errors is usually minimised, since while the

design process advances, the more complex the design gets, and the more effort is needed if every detail is compared to specification. The verification is easier, since it is enough to verify that there have not been alterations in behaviour compared to the preceding phase product. The effect of verifying each phase product is that the final product is actually validated since if all the phase products conform to the preceding level and the earliest phase product is the specification, the end product conforms to the specification.

The validation in co-design workflow can be divided into three different groups: behavioural validation, co-validation and co-verification. Behavioural validation is used for validating the specification of the system. Validation may be achieved by simulation or formal model checking. The idea is to reveal any logical faults, such as deadlocks and unwanted state transition sequences. In the behavioural validation, however, the reference is the paper version of the specification, not the existing executable specification, thus the validating process requires lots of designer involvement. Co-validation takes place after the system is partitioned into parts to be implemented with different technologies. The idea is to validate that the system still behaves as specified, even though it is constructed from several individual pieces of designs. Co-verification is intended to use design consistency throughout the design process by allowing design consisting of different technologies to be verified against the descriptions of earlier abstraction levels, thus effectively causing the end-product to be validated. The term abstraction level describes the top-down strategy, where we start from the highest level, general system lay-out and end with the lowest level, where individual transistors are described in detail [3, 5, 6].

2.3 CO-VERIFICATION METHODS

Validation of mixed software and hardware is expensive and time consuming. Usage of current techniques in complex system-on-silicon products delays the release of products or leads to insufficient validation. If, however, the product is inadequately validated, the problems are only moved to the end product, where the correcting of an error is most expensive.

Currently, software and hardware are developed separately and then integrated together. This integration is usually time consuming, since there are often problems and errors which are not revealed before the integration. Some of the errors can be corrected easily by modifying software, which may lead to a reduction of the functionality of the end product in some degree. In the worst case, however, the correction of error may require re-designing of hardware, which requires time and causes additional expenses.

In order to reveal errors, the hardware and the software components of a complex embedded system need to be tested as a whole. Preferably this should be possible even during the development, i.e. it should be possible to test the hardware and software against each other before either of them is ready.

During the early phases of the design process, a graphical specification and/or design languages may be used to construct executable specifications of the systems behaviour. The executable specifications can be used for behavioural validation of the system, but they are not capable of verification in lower abstraction levels.

Hardware simulators allow the hardware to be verified and also the software to be executed on the simulated hardware, which enables co-verification of the phase product. Most of the HDL simulators today are event-driven, meaning that the simulation is based on events happening at arbitrary times. This method makes possible simulation of both kinds of logic circuits, synchronous and asynchronous. However, most of the circuits found from embedded systems are not asynchronous, which makes possible the use of clock cycle boundaries as a basis for simulation execution. This type of simulation is called cycle-based. The benefit from cycle-based simulation is the reduction of scheduling events, which speed up the simulation over 10 times compared to event driven simulation. The problem with both types of HDL simulations is that when all the details of the hardware are modelled, the simulation, although exact, is very slow. Running software on a design in a simulated hardware leads typically to a performance of 1 to 10 instructions (in cycle based 10 times more) per second, depending on the system's complexity and simulator performance. This speed is enough for executing small portions of the software part of the system under design, like hardware drivers, but rules out the testing the whole software partitioning of the system [7, 8, 9].

It is, however, possible to speed up hardware simulation by using hardware accelerators, which are specialised computers, allowing the gate-level HDL description to be executed faster than it would be with an ordinary workstation. Typically these accelerators allow simulation speeds of thousands of simulated clock cycles per second, or millions of hardware events per second. The model executed in hardware accelerators is gate-level description, which causes two types of problems. On the one hand, gate-level model is very low level description, thus disabling the possibility of simulating early prototypes. On the other hand, the processor-vendors do not usually provide gate-level models, since they are far too detailed and implementation specific, and thus cause a danger of intellectual property leakage.

The hardware accelerators cost typically several hundred thousand dollars, which usually rules out accelerators for ordinary co-verification purposes. If, however, a company already have a hardware accelerator, and if it is easy to integrate a software debugger into it, the hardware accelerator can be a useful alternative in co-verification [10, 11, 12].

Another hardware assisted simulation method is the FPGA emulator. The FPGA emulator includes a Field Programmable Gate Array (FPGA) device, which enables the logic functions of target ASIC to be programmed to FPGA. FPGA can then be used as if it was the target ASIC enabling real in-circuit-emulation. The FPGA is fast, 1% to 10% of real designs execution speed, but it does not suit complex systems well since the design must be divided into blocks of 600 to 2000 gates. The small size of the block is caused mainly by limited I/O capabilities of the FPGA device, and thus poor utility of the FPGA's capacity when connecting several devices together. The problems with FPGA devices is mainly the same as with the hardware accelerators and additionally the poor visibility to design, i.e. the inner state of the design during execution. Also the compilation of the system into the FPGA is not always very straightforward, which causes additional overhead in the development of a simulation model to the system [9, 12, 13].

The hardware emulator is aimed at overcoming problems related to hardware accelerators and the FPGA emulator. Basically it is a set of ASIC circuits which allow accelerated simulation of the VHDL design. The speed of the simulation is at the same level as in FPGA emulators, but the visibility to design and the

compilation speed are better. The main disadvantage is the cost of the emulator, particularly if large designs are simulated [10].

The slowest hardware assisted method is hardware modelling. This method features a real processor executing the software and a software simulator running the models of the peripheral and custom circuits. The main advantage over previous hardware assisted methods is that simulation does not require a processor model that may be impossible to obtain. The overall speed of the simulation set-up is roughly the same as with HDL simulators, which makes hardware modelling suitable only for small scale software execution [9].

If the system consists of a standard processor core, the execution performance of software part of the system may be increased by using software simulators. Software simulators enable the execution of target processors' code in the host workstation. There are basically two different solutions for software simulation, an Instruction Set Simulator (ISS) and an operating system level simulator. In the instruction set simulation, all the instructions of the target processor are interpreted to host workstation instructions, thus enabling the execution of the target code. ISS's are typically accurate, since the whole code is interpreted in machine instruction granularity. The drawback is the performance, an execution rate of well under 1:200 host instructions is required to execute one target instruction i.e. if the host executes 200 MHz, the target code executes below 1 MHz. Also the possibilities to model custom hardware are minimal or require considerable modelling effort. These simulators are commercially available for many different processors, possibly even before the target processor exists, since processor manufacturers use ISS's in their own processor development [14].

If the operating system of the target environment is modelled in the host workstation and the target code is then compiled to be host executable and executed in the target operating system, fast simulation is achieved. Interpretation of the instructions is not required any more, which enables the target code to be executed as fast as the host code. However, this approach requires that the application is written in some portable language that enables the code to be compiled in to the host environment, thus causing additional modelling effort. The major drawback compared to ISS is that the accuracy and

timing information in code execution is sacrificed. Also the behaviour of the hardware is difficult to model [14].

The oldest and probably most reliable way of verifying the mixed HW/SW systems behaviour is a hardware prototype. However, the system-on-silicon trend is not in line with this method, since the prototype of an ASIC is expensive and it takes too long to manufacture. The ASIC prototype is also complex to debug, since the inner functions of the ASIC are not visible to the designer.

Since no single co-verification method provides a satisfactory result, the validation problems have to be solved by combining several methods. Co-simulation makes it possible to combine the software emulator(s) executing the target core processors software in the host workstation, simultaneously with custom hardware simulated in HDL simulators (maybe boosted with hardware accelerators). This method makes use of the fact that the whole system is rarely based totally on custom hardware, but uses standard core processors and Digital Signal Processors (DSP), which usually have emulators available. The co-simulation can provide good visibility and debug features in the system behaviour and simultaneously enable the software and hardware to be tested against each other, before either really exists. The greatest difficulty related to co-simulation is the connection of simulators in an efficient way without sacrificing too much accuracy.

3. CO-SIMULATION

There are basically two major approaches in the implementation techniques of co-simulation: heterogeneous prototyping and homogeneous prototyping.

Homogeneous prototyping means that the different parts of the system are first described in common description language, and the description is then simulated with one or several simulators using the same description language. The structure of the homogeneous approach is depicted on the left side of Figure 1. The homogeneous approach can be used in simulation of specification and also in partitioned design where the mapping between software and hardware is done. When the design process advances towards implementation, more task specialised languages, such as VHDL and C are needed. However, moving from common description format to specialised languages is not a trivial task. The homogeneous prototyping approach, although providing interesting possibilities, suffers from the non-existence of a common description language. This reduces the possibilities of reuse of design, and slows down the development of automated compilation from common specification language to lower level languages [15].

A heterogeneous prototype is an executable system model in which different parts may be specified at different abstraction levels, and yet they can be executed together as one system [16]. If this definition is extended to parts of the system that are described in different description languages, we get the definition for the heterogeneous co-simulation. In this document, the term co-simulation means heterogeneous co-simulation. The structure of the heterogeneous approach is depicted on the right side of Figure 1.

The advantages of the heterogeneous approach are that it does not require the use of new simulators or design conventions, and it eases the usage of the existing designs as components in the construction of a new design. The drawback in the heterogeneous approach is difficulties in constructing the connection and synchronisation mechanisms between different simulators.

The possibility of selecting a suitable simulator type, depending on the description language and/or abstraction level, boosts the speed in heterogeneous co-simulation. The other possibility of improving performance is to simulate the

existing and already validated part of the system in less detail than the parts that consists new custom hardware. This is possible since most software intensive embedded systems today utilise complex software running in a standard core processor, it is faster to simulate this software execution in a standard instruction set in less detail than the custom hardware. However, if the detailed behaviour and timing of the custom hardware are needed, more accurate HDL simulation is required. This leads to a solution where most of the execution is done in the fast software simulator, whereas interesting interactions with the custom hardware are simulated in a slow but accurate HDL simulator.

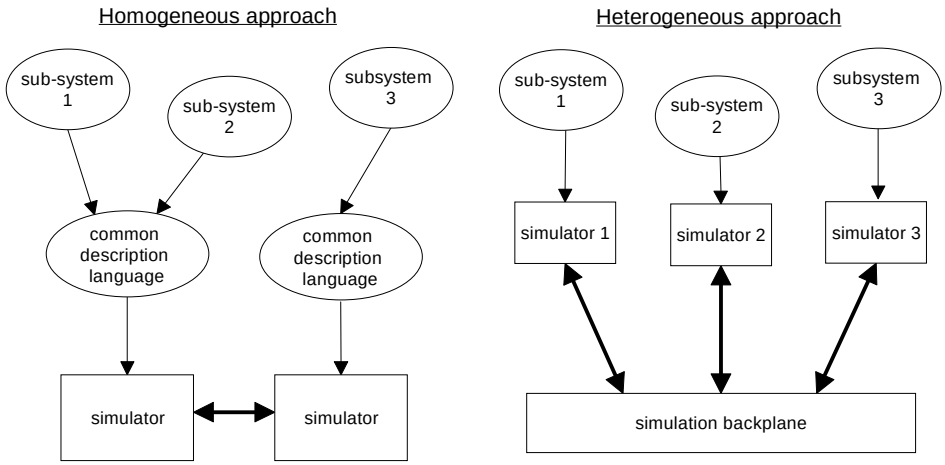


Figure 1. Differences in the homogeneous and heterogeneous approach.

The amount of the system's functionality moved from an HDL simulator to a software simulator may vary a lot, depending on the required accuracy and the structure of the system. For example, by moving the memory references, such as reading and writing the data, from an HDL simulator to a software simulator can yield significant improvement in performance. This, however, requires that the bus activity of the system is not particularly interesting from the hardware point of view, since the memory references made in software do not excite the hardware simulation.

3.1 IMPLEMENTATION OF HETEROGENEOUS CO-SIMULATION

There are two basic ways of connecting simulators: a case dependent custom linkage and a simulation backplane.

Custom linkage is widely used in co-simulation research and in some ad-hoc solutions. The advantage of this solution is that the communication protocol required to allow two pre-defined simulators to communicate and to synchronise with each other is simple. Simplicity allows the fast and cheap development of a simulation environment, and a custom solution can also make use of the simulator's special shortcuts in order to attain better performance.

The simulation backplane is a more general solution, providing synchronisation and communication between simulators. The backplane can be a function library or separate tool. The advantage of simulation backplanes over custom linkage is the flexibility. As simulators develop and their performance increases, updating of the environment is needed. In the custom solution, updating one simulator could lead to redesigning of the whole simulation environment again. In the backplane solution, only the interface of the new simulator should be developed and the rest of the simulation environment remain unchanged. A well-defined interface to the backplane also enables the development of an application interface in the form of a function library to ease the job of connecting new simulators.

Another reason favouring the backplane solution is that several simulator types can be used in different stages of design. For example, existing VHDL models may be first simulated with simulatable specification and then later with a cross compiled version of software and then finally with instruction set simulator executing the final version of the software. In a custom solution this would require several co-simulation environments (VHDL and specification, VHDL and compiled code, and VHDL and an instruction set simulator) but in the backplane solution, the same VHDL simulator can be used and only the other simulator is changed.

From the simulation backplane, allowing two simulators to be connected to each other is only a short way to develop a backplane that allows co-simulation of several simulators. Most of the additional work is related to synchronisation of

several simulators, which requires more complex algorithms than with the case of two simulators. The advantage of having several simulators is that the system under simulation may consist of several different descriptions at mixed abstraction levels. For example, a simulation set-up may consist of one vendor provided VHDL model of a hardware component, a behavioural description of a DSP processor, an instruction set simulation of software, a simulated specification, a test bench, etc. The additional value of having several simulators is that workload, otherwise concentrated on one or two workstations, can be divided between several workstations allowing much more processor power to be used in the simulation of one system.

3.2 RELATED WORK

Co-simulation environments are currently being developed by several companies and research organisations. The problem with current solutions is the limited speed, typically less than 5000 instructions per second. The speed varies a lot depending on the actual system under design, since increasing hardware interactions slow down the simulation. The timing accuracy of the provided solutions depends on the desired speed, while more speed typically means less accuracy and vice versa.

EDA-vendors provide the simulation backplanes allowing HDL simulators to be connected to software or specification simulation. The other solution used is to relieve the burden of processor emulation from the HDL simulator using an add on software processor emulator. Typically these solutions have a limited amount of supported processors or/and simulators.

The Yokogawa Electric Corporation provides a Virtual In Circuit Emulator (VirtualICE) which enables the concurrent simulation of hardware and software parts of a system. VirtualICE is based on co-simulation of a VirtualICE CPU/DSP module providing target processors functionality and Verilog or a bilingual simulator. Additionally, VirtualICE allows the designer to view and alter the contents of ROM, RAM and processor registers, just as in the real in circuit emulation. The software part of the system under simulation is located in the system’s memory modelled in the Verilog simulator [17].

Ptolemy is a research project conducted by the DSP Design Group of the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley. The main application area is digital signal processing, but Ptolemy can be used for other applications as well. Ptolemy provides a software environment for specifying, generating and simulating code. In the future, the environment will provide interfacing abilities to existing design and simulation environments [18].

The Seamless Co-Verification Environment (Seamless CVE) is the Mentor Graphics' solution for hardware/software co-simulation. Seamless CVE enables Mentor Graphics' logic and HDL simulators and a Microtec XRAY software simulator to be connected. Seamless CVE enables also user-controlled optimisations to boost performance by isolating the logic simulator from software intensive operations such as block memory transfers and algorithmic routines. Seamless CVE supports also the SimExpress hardware emulator, which speeds up the overall performance even more [19].

Viewlogic Systems Inc. provides the Eagle I and Eagle V co-simulation tools. The Eagle V is intended for ASIC validation and Eagle I for validating embedded systems. Both of the Eagles useVirtual Software Processors to emulate standard processor cores, which reduces the task of the HDL simulator, thus allowing a faster simulation. Eagle I allows the integration of both software development environments and hardware environments, whereas Eagle V is mostly aimed at simulation of hardware systems, although it does have limited capabilities of software execution. Eagle products support numerous HDL simulators and software development environments [20].

CoWare Inc. has a solution for the development of mixed hardware and software systems. CoWare enables the animation of system specifications written in existing languages such as C, VHDL and Verilog. CoWare features automatic HW/SW interface generation, optimisation and analysis of partitioning between HW and SW. In the co-simulation the CoWare enables the software compiled to a target processor to be executed and debugged concurrently with a virtual prototype of hardware [21].

VSIA is an organisation that includes representatives from all segments of the system-chip industry. VSIA does not provide any tools for co-simulation

environment development, but it does concentrate on solving the problems of Intellectual Property (IP) questions related to the development of systems on silicon. The problem with current system level co-simulation is the lack of components, CPU's,DSP's etc., available as simulatable pieces of design. The companies involved with HW design do not want to provide accurate models of their designs, due the risk of IP leakage. The VSI Alliance is trying to standardise the "Virtual Socket Interface" which would enable the secure encapsulation of the pieces of design into simulatable black-boxes, allowing exact simulation, but providing no hints of the actual design [22].

Quickturn Design Systems Inc., better known for their FPGA emulators, provide a Q/Bridge interface that allows integration of HDL and Instruction Set Simulators (ISS) of other Electronic Design Automation (EDA) companies into their emulation environments and hardware accelerators [13].

The Cadence Alta Group provides system-level design environments where VHDL or Verilog code can be linked to co-simulation with system behaviour simulation. The system behaviour simulation is based on processes written in C-language and executed on a Signal Processing Work System(SPW) by using the tool called the Convergence Simulation Architecture processor. Additionally SPW can be used in co-simulation together with BONeS DESIGNER, which models and analyses the systems protocol/messaging layers [23].

3.3 INDUSTRIAL NEEDS

One of the biggest bottlenecks in today's electronic design is not the lack of development time. It is usually acceptable to allocate large amounts of resources in the right place, but if the whole project is waiting for some part of the project to be ready before the next phase can begin, the project has major problems. This has caused companies to be cautious in adopting new design methods in to their fine-tuned design processes, since if not successful, the new methods and tools may endanger the success of whole project.

However, companies need new tools in order to enable longer simulations in a shorter time to verify more complex systems than before. For example, verifying a user interactive program that controls some custom hardware may require user

interactive testing in order to achieve accurate test results. Testing such systems requires execution of several millions or billions of instructions in order to cover even a fraction of the states the system may end up in. This kind of testing, in order to be done at all, requires that the execution speed of software in the co-simulation environment should be at least 10^5 instructions per second and if real interactive testing is required an execution rate of at least 10^6 would be needed. Such figures are impossible to achieve with the current simulators, and new simulation methods are needed. A typical Instruction Set Simulator (ISS) today can run as high as 10^5 instructions per second, but connecting it to co-simulation typically worsens the figures to 10^3. However, since the companies are not willing to make large alterations in their design conventions, a system that would allow the existing tools to be used as they are usually used, but simultaneously enabling fast simulation of the software part would be required.

During the OMI/TOOLS project, a questionnaire was sent to 14 companies in Finland, Germany and Austria. The questionnaire gathered the requirements for a new co-simulation and modelling environment to be developed in the OMI/TOOLS project [24].

The profile of the respondents varied from basic designer to upper management. The most common duty was a designer, responsible for developing the specification and design for a new SW/HW system. However, the integration, system analysis and implementation were also very common duties. This means that most of the answers were given by persons who actually are directly involved in product development [24].

The following issues were considered to be the most important in the answers:

- software testing before hardware is available. This notion is one of the basic reasons for co-simulation, and therefore it was not a surprise that it mentioned in nearly all of the answers.

- performance of 10^6 instructions per second, which is much more than can be achieved today. Not a surprising result though, since longer simulations are needed to simulate the

more complex systems. Still time consumed should not increase but rather decrease compared with time used today.

- flexibility i.e. possibility to add new tools to the environment and customise the environment as desired

- design reuse by using existing designs as a part or the basis for the new design. Graphical Model Builder (GMB) and Model DataBase (MDB) support design reuse, by componentiation of parts of the design.

- accuracy and good match for target in simulation. An accuracy better than 1% was desired, which is more than typical software simulations can offer. However, a good match for target is essential, since it reduces the need for verification in later phase products and with the final product, thus decreasing the time to market.

- usage of the new environment as a part of the current design process. Companies involved in the electronics industry have usually their own design processes which they do not want to change at once, but rather develop constantly and gradually.

- maintaining the performance of simulator components

- reuse of existing tools

- integration of the VHDL simulator to MSE. VHDL is the most important Hardware Description Language used in Europe and designers are used to working with it.

- hierarchical model building facilities.

4. OMI/TOOLS ENVIRONMENT

The tools created in the OMI/TOOLS project will enable early and on-going verification of mixed hardware/software designs between specifications. This will allow the errors to be detected at the early stages of design, thus reducing costly errors in the later design stages and final product. The OMI/TOOLS project features horizontally and vertically integrated methods in the hardware and software design process as seen on Figure 2 [25].

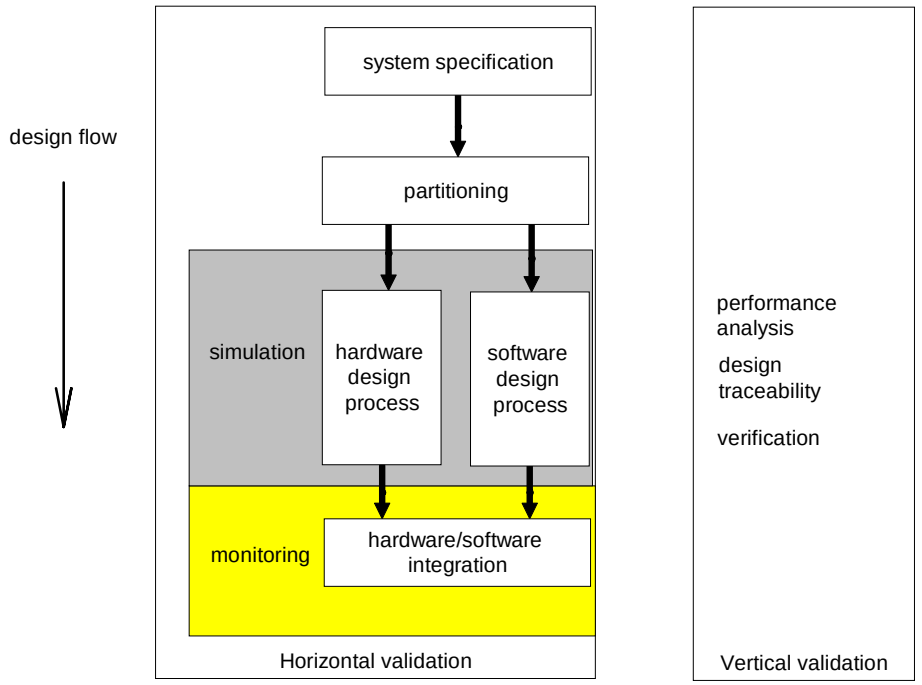


Figure 2. Horizontal and vertical validation in an integrated design process.

Horizontal validation in the OMI/TOOLS support the validation of hardware/software systems in different abstraction levels. A mixed-level simulation environment allows the simulation and visualisation of behavioural, execution or instruction-driven modules together with operating system models. Hardware/software monitoring is possible with real or simulated systems.

Vertical validation is for maintaining the consistency between the different phase products. This is achieved by performance analysis, formal verification and traceability of the design history.

VTT Electronics from Finland and the German company SIBET are responsible for the development of the Modelling and Simulation Environment (MSE). The MSE is an environment which features graphical model building and mixed hardware/software simulation at different levels of abstraction. Additionally the MSE allows simulation to be visualised by graphical animation.

4.1 MODELLING AND SIMULATION ENVIRONMENT OVERVIEW

The Modelling and Simulation Environment (MSE) of the OMI/TOOLS project is an open development and verification environment for embedded control systems. MSE enables the design and simulation of mixed hardware and software systems. The basic idea behind the MSE is to enable integration of the existing simulators and model builders so that descriptions written for different modelling techniques can be modelled and simulated together.

MSE itself does not specify the abstraction level of the system under simulation. This approach makes it possible to use MSE during different phases of a design process, for example, for validation of specification, system architecture design or implementation design. It is even possible to mix models described at different abstraction levels in one system under simulation, and verify the specification of the hardware against an existing software part or vice versa.

MSE is designed to be easily expandable with new simulators or other tools. MSE encapsulates parts of the system under design into components that include the interface and functionality. The functionality is aimed only at the target simulator, which enables the existing modelling techniques and models to be used unaltered for a generation of new design components. However, a separate interface (entity) makes possible the use of common, simulator independent, modelling conventions for system level modelling. This makes the MSE modelling easier to learn since the designer does not need to learn the modelling techniques used by the simulators connected. The standardised

interface allows the simulation of components written in different description languages.

4.2 COMPONENTS OF THE MSE

The MSE consists of a graphical modelling environment called Graphical Model Builder (GMB), the simulation backplane InterOperation Engine (IOE), the animation tool Graphical Animator (GA). Currently two simulators are available for MSE, one for software simulations and one for hardware simulations. The different parts of the MSE share data stored in the Model DataBase (MDB), which enables information exchange during modelling and serves also as a storage for the components and systems under design.

The structure of the MSE is depicted in Figure 3. The MDB is described as a Modelling integration backplane and the IOE as Simulation Integration backplane. The figure is strongly simplified but it does reveal the overall structure of the MSE.

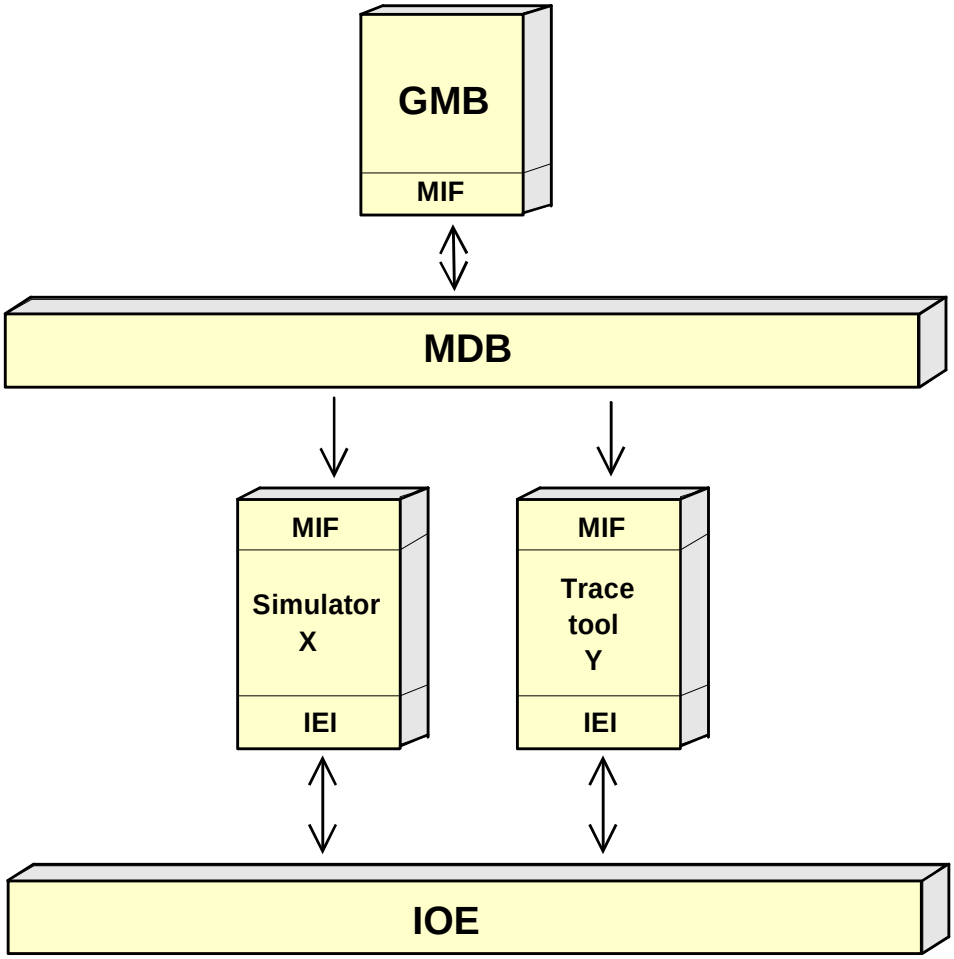


Figure 3. The components of the MSE.

4.2.1 Model Database

Component re-use requires that components are easily available and accessible. The components are stored in a database where the data is readily available and the consistency of the data can be verified. This is especially important when the amount of different components and the tools accessing them increases. Additionally the database can be used for sharing the information between different tools connected into MSE and also storing the final systems for simulation. The storing of complete systems enables the designer to experiment

with different variations of the same system without losing the old and possibly functioning version.

The MSE Model DataBase (MDB) is based on a commercial SQL engine. The database engine selected by VTT is the Hughes Technologies Mini SQL, which is a lightweight SQL database engine supporting a subset of ANSI SQL specification. However the MDB itself is compatible with any SQL database engine [26, 27].

The MSE is accessed via a function library called the Model Interface API (MIF API), which enables easier integration of MDB access to new tools. The structure of MDB access is depicted in Figure 4.

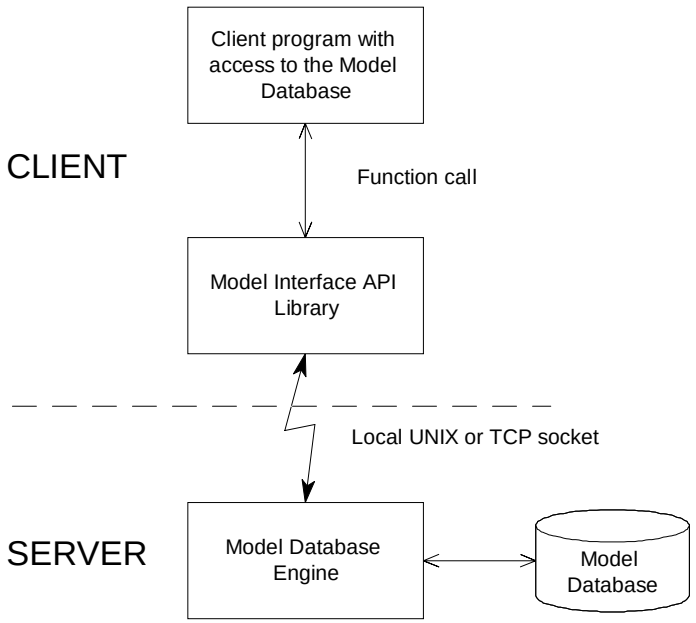


Figure 4. The MDB access of the application in MSE.

Individual design entities (components), design modules, are used for constructing the system under simulation. The design module is a stand alone piece of design which can be simulated by its target simulator. The design module consists of a design module file and an interface specification. The design module file includes the information for the target simulator to load and

simulate the design module. The design module file does not contain any information from the modelling point of view, and it is meaningless to all other tools except to the target simulator. The interface information, i.e. flow entries, enables a design module to be connected to other design modules. The interface information is simulation technology independent, which enables the common modelling conventions to be used, and therefore the design modules described in different languages may be interfaced.

The system under simulation is described in the system model, which consists of several instances of design modules and their interconnections. Design module instances in a system model are called objects. When the system model is created, the objects selected into system are connected to each other, and the connection information is then stored into the MDB. Although the system model describes the whole system under design, it does not contain information on the simulation. In order for the system to be simulated a simulation model must be generated. The simulation model is basically a system model where objects are allocated to simulators, and the connections are allocated to tracers i.e. tools that gather the communication information. One system model may have several different simulation models, which enables the system to be simulated in different simulation environments with different sets of simulators and enables the designer to balance the load between different instances of the same simulator. Several simulation models also enable the designer to concentrate only on the signals of interest to reduce the tracing load.

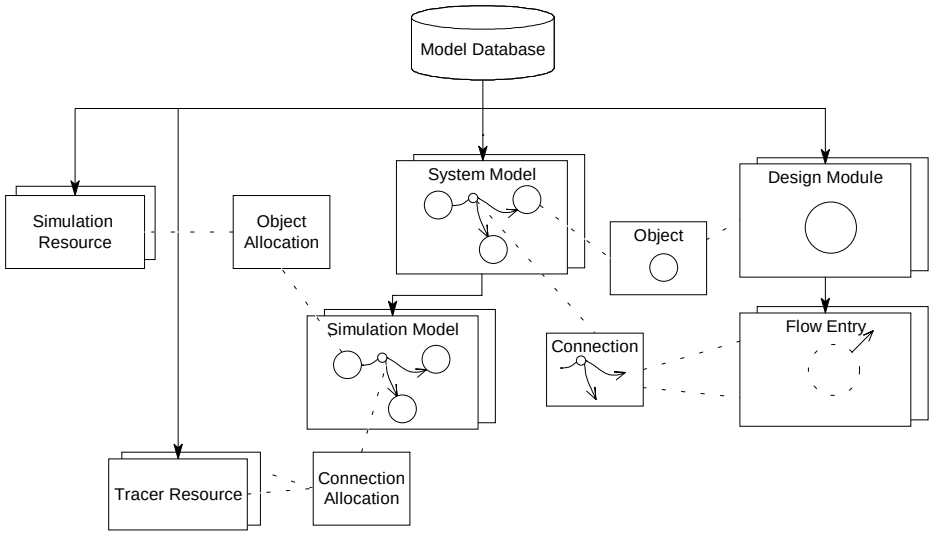


Figure 5. The structure of the MDB.

Figure 5 illustrates the main structure of the MDB. The dotted lines describe the dependencies, and the arrows describe the hierarchical order, i.e. a design module contains the flow entries (interface) and a system model contains simulation models [27].

4.2.2 Graphical Model Builder

A Graphical Model Builder (GMB) is used for describing the system under design and the system under simulation. The commercial Prosa^{TM} SASD tool is used as a graphical system model editor [28]. Other parts of the GMB generate and interpret the graphical diagrams used by Prosa and manage the system and simulation models. Additionally, the GMB allows the designer to select system and simulation models for simulation and to start the simulation.

Figure 6 illustrates the structure of the GMB. The leftmost branch represents design module editing, the middle branch system model editing and the rightmost branch simulation model editing. The leftmost greyed box represents the design module editing, where new design modules are stored in to the MDB. Storing a design module can be considered as registering a design component for future use in the system design. The greyed box in Figure 6 may be replaced

by individual model builders intended for building models for specific types of simulators in the MSE (for example a VHDL model builder).

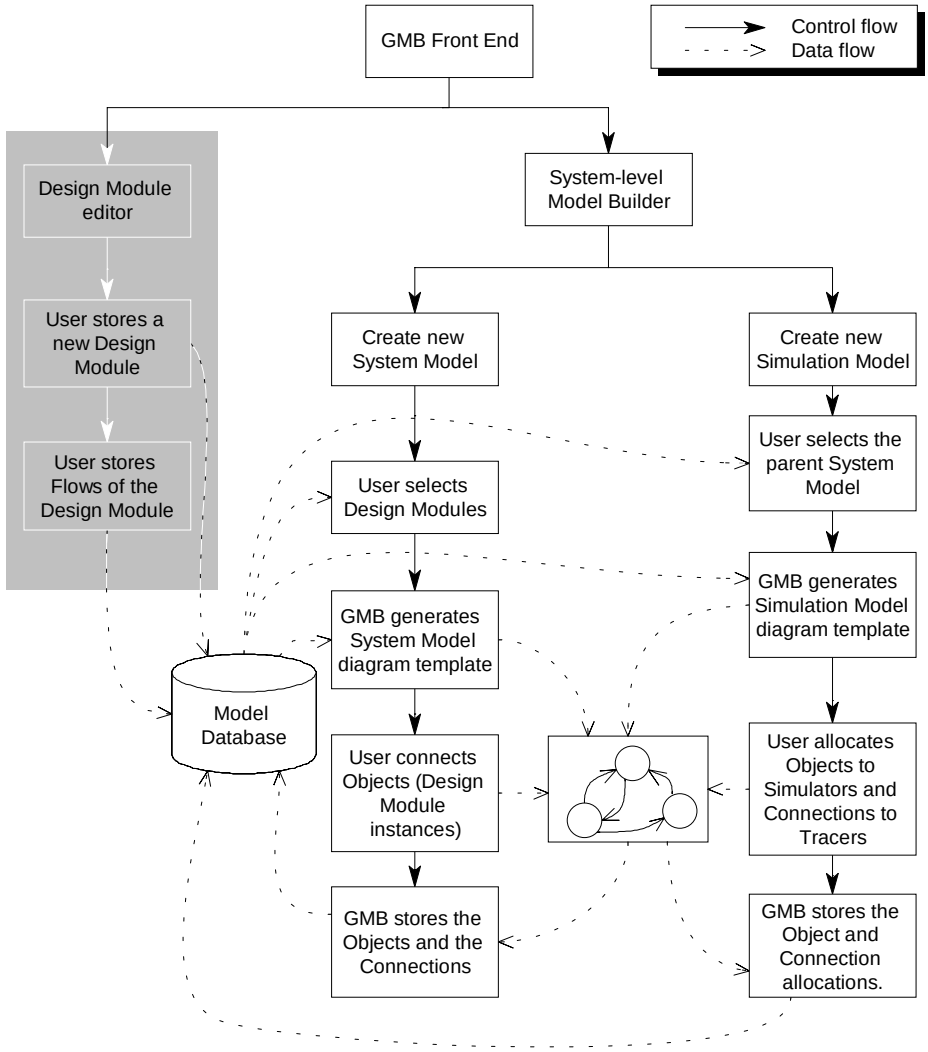


Figure 6. The structure of the GMB.

The workflow of the model building can be divided in to three parts: building a design module, system modelling and simulation modelling. Building of a design module takes place after the design component has been created and tested and can therefore be regarded as a prerequisite task for the system-level

modelling. Other mandatory information for system modelling is that the MDB contains the information on the simulators that may be used and the data types supported.

Building of a system-level model consists of system modelling and simulation modelling. The middle branch of Figure 6 describes the system modelling. In system modelling, the designer names a new system model, selects the design modules that form a system, and connects the design modules together with the SASD editor. Finally, the GMB stores the system model into the MDB. A simulation model is derived from the system model. One system model may have multiple simulation models. In simulation model building, the designer selects a system model, that acts as parent to the simulation model, allocates the objects for the appropriate simulators and sets the connections of interest to be traced by one or more tracers. After the simulation model is created, the GMB executes the Simulation Configuration Generators (SCGs) for every simulator involved in the simulation model. SCGs are simulator wise tools, that prepare a piece of the system to be simulated by its target simulator. For example, for QuickHDL VHDL simulator, an SCG composes the objects allocated for the simulator into one composition object which can then be loaded into the simulator. The process of simulation modelling is illustrated in the rightmost branch of Figure 6.

4.2.3 Inter Operation Engine

The Inter Operation Engine (IOE) is a co-ordinator of the messages between different tools involved in the simulation. The IOE maintains also the logical simulation control by synchronising the simulators. A simulation session is divided into several phases. Every tool must successfully complete current phase, before the IOE allows the tools to proceed into the next phase. This way one makes sure that all the tools involved in the simulation set-up have accepted the initialisation messages before the simulation session sequence is continued.

Before a simulation session is started, communication channels between IOE and simulators must be established. The IOE starts first every tool involved in the simulation set-up, and then waits for the tools indications that they have started. When a tool is running, the IOE establishes a socket connection to it. When all the tools are running, the IOE proceeds to the initialisation phase. In

the initialisation phase the IOE and the connected tool agree upon the connection and the protocol version used. After initialisation, the IOE proceeds to the session set-up phase. During session set-up phase the IOE and the connected tools exchanges the data of the current session: the tools involved, the connections between the tools and the design name. After this phase each tool in the simulation set-up is ready for the simulation. The simulation begins when all the connected tools have indicated that they are ready to simulate. IOE controls the simulation by sending different kinds of stepping commands to simulators. The IOE computes the length and the type of step, based on the next event time message received from each simulator. When a simulator has a signal transaction in its external output signal, the simulator constructs and sends an event message to the IOE. The IOE re-routes the event message to the target simulator, and if the signal is marked to be traced, the message is also copied to the tracer.

For efficient communication between IOE and tools, a communication protocol is constructed. The protocol is called the InterOperation Engine Communication Transmission Protocol (ICTP). The ICTP protocol is a message based client server protocol where the IOE is a server and all the other tools are clients. The ICTP messages are constructed by calling functions in an IOE/IEI Communication Library (ICLib). However, the sequence of the messages is still the responsibility of the application [29]. The ICLib includes functions for different stages of inter tool communication, like handling the communication channel, constructing and sending messages to the other tools and reading messages from them. The communication layers involved in a two simulator set-up are depicted in Figure 7 [30].

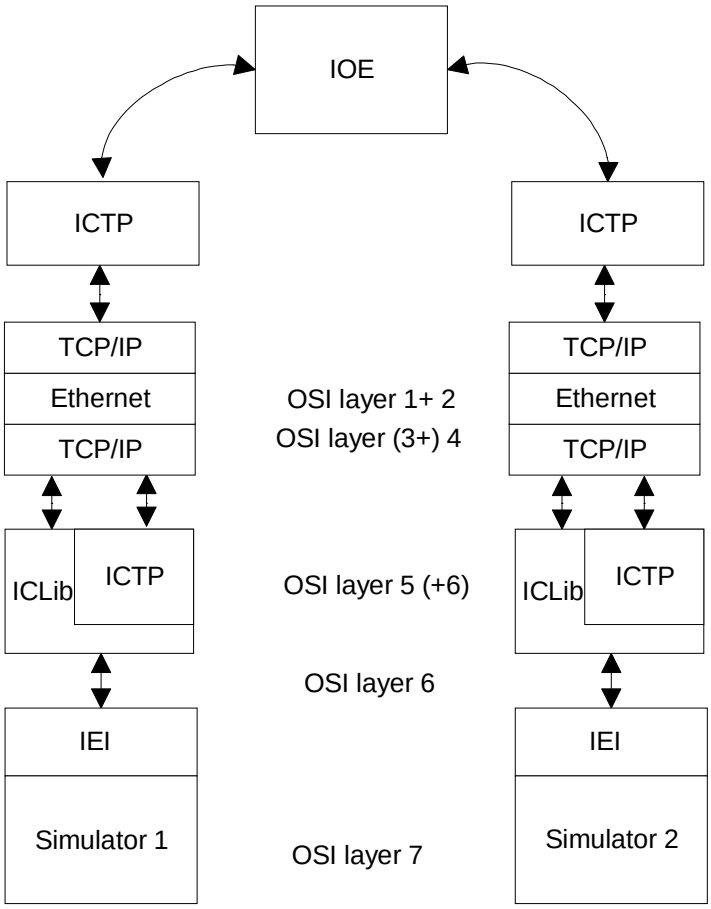


Figure 7. The structure of the inter simulator communication.

Data types

The IOE transfers the simulation data between simulators as event messages. An event message consists of the time value of the occurrence of the event and a value to be transferred. For routing the signal to the correct simulator, the event message contains also a signal identification number. Types of pre-defined event messages are described in Table 1.

Table 1. The event types of ICTP protocol.

Message name	Event Value
ICTP_BIT_EVENT_MSG	bit event
ICTP_CHAR_EVENT_MSG	signed byte event
ICTP_BYTE_EVENT_MSG	unsigned byte event
ICTP_INT16_EVENT_MSG	signed integer of 16 bits event
ICTP_UINT16_EVENT_MSG	unsigned integer of 16 bits event
ICTP_INT32_EVENT_MSG	signed integer of 32 bits event
ICTP_UINT32_EVENT_MSG	unsigned integer of 32 bits event
ICTP_REAL_EVENT_MSG	real event
ICTP_DOUBLE_EVENT_MSG	double event
ICTP_TIME_EVENT_MSG	time event
ICTP_UNKNOWN_BIN_EVENT_MSG	binary event of unknown type
ICTP_UNKNOWN_TEXT_EVENT_MSG	text event of unknown type

The InteroperationEngine Interface (IEI) of the simulator, on receiving the event, checks if the event type can be accepted. The ICTP protocol or the IOE does not care about the types of events it transfers. If the IEI cannot insert the event value into the simulated model, an error message is generated. This approach allows the simulator, which can only generate, for example, 16 bit integer events, to be connected to a VHDL simulator simulating model with byte type signals. The only requirement is that the VHDL simulator's IEI can map the 16 bit event into a byte type signal. The ICTP protocol supports also the transfer of more complex events. The ICTP_UNKNOWN_BIN_EVENT_MSG and ICTP_UNKNOWN_TEXT_EVENT_MSG events can be used if the sending and the receiving IEI's agree on the meaning of the event data. However, there is no common way to interpret these event types.

Synchronisation

During the simulation run, the IOE maintains the synchronisation by allowing the simulators to be run only in steps. This means that the simulation is effectively divided into several sub-phases. The length of the step may vary depending of the simulation type and the state of the system under simulation.

The simulation sub-phase is considered to be safe i.e. no incoming signal or control message may not effect the simulation of the sub-phase after the simulator is entered in to it. This requires that the IOE must have an ability to compute the safe length of the sub-phase. The IOE needs to know the next event times of all the simulators in order to compute the length of the next sub-phase. The next event time is an absolute safe time, and there cannot occur any event in any condition between current time and next event time.

4.2.4 QuickHDL

Mentor Graphics' QuickHDL is a modern HDL simulation environment featuring the source, structure, process, dataflow, wave, list, signals, and variables windows for design analysis and debugging. QuickHDL fulfils the IEEE-1076-87 and IEEE-1076-93 (VHDL), IEEE-1364 (Verilog) standards and enables both VHDL and Verilog to be used simultaneously in the same design. However, in the OMI/TOOLS project, only the VHDL simulating capability is used. QuickHDL provides an IEEE-1076-93 standard foreign language interface, which is implemented as a C-function library. The behaviour described in C-language is compiled with the host computer's C-compiler to an object file or library. During the simulation start, the foreign processes as well as the VHDL and the Verilog processes, are linked to QuickHDL and executed.

4.2.5 ClearSim

ClearSim 1 is the first version of the software simulator used in the MSE. It is a result of the OMI/CLEAR project and it has been developed into a more advanced ClearSim 2 in the OMI/TOOLS project. The MSE ClearSim enables host-based simulation of the software parts of the systems. ClearSim comes with the models of a microprocessor core and a real time operating system for enabling the software validation. The parts of the systems to be simulated with

ClearSim are described in C-language. ClearSim uses execution-driven simulation, which means that the target processor's application code is instrumented to contain the timing information, and then it executed in the host workstation. The main difference when compared to the Instruction Set Simulation (ISS) is that in the ISS, the target instructions are interpreted one by one, but in ClearSim the target code is compiled into a host executable form. The starting sequence of the ClearSim is described in Figure 8.

The execution time of the target code is estimated on the instruction level such that the time consumed by every instruction is considered separately. Then the target instructions are compiled into host assembly code and the timing information is added to the assembly code. However, after the timing of each instruction is estimated, small blocks of several instructions are formed and the blocks' timing information is calculated by taking the sum of the execution times of each member of a block. The blocks are formed because they reduce the amount of control and timing points in the instrumented code and thus allow faster execution. Finally, the host assembly-code with the control hooks are compiled into host a object-file.

In order to execute the simulation, the object file and the instrumented version of the operating system are linked together with a ready compiled simulation kernel. The result is a process image which is loaded and started during the simulator startup.

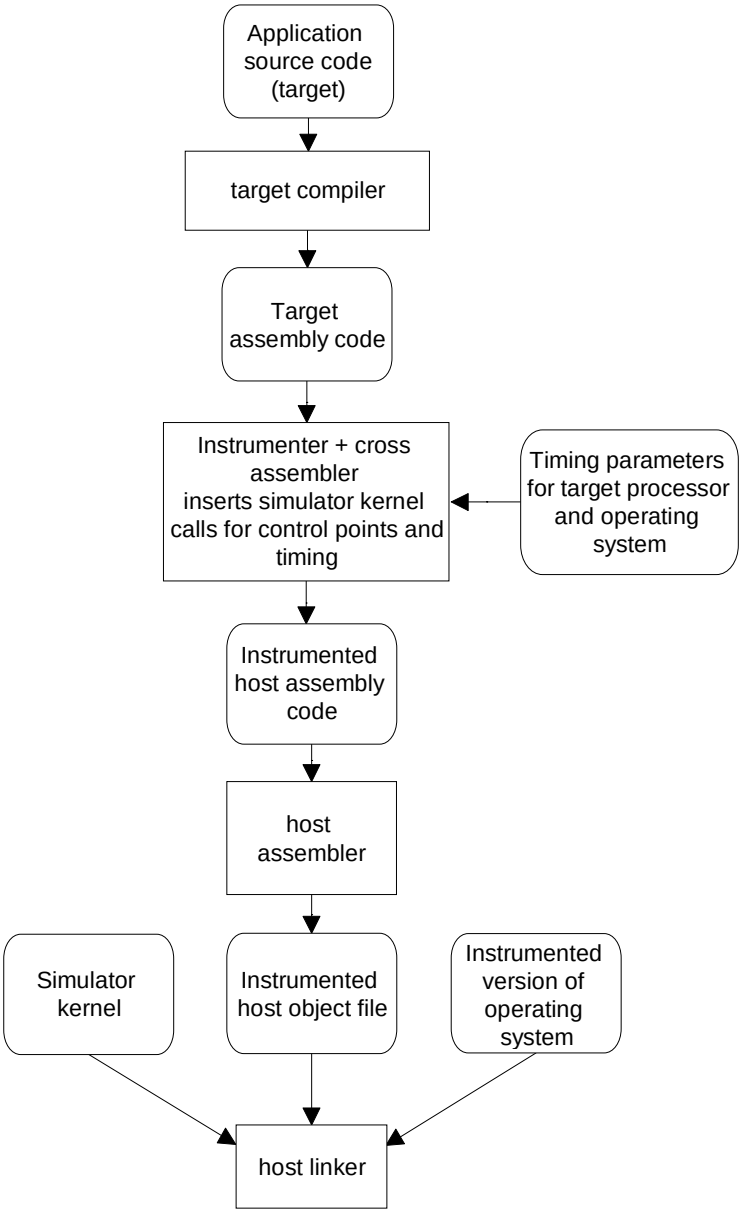


Figure 8. The starting sequence of the ClearSim.

4.2.6 Graphical Animator

The Graphical Animator (GA) in the MSE provides a high-level view of the system under simulation and overall status of the simulation. Visualisation is an animated trace of the messages exchanged between subsystems of the whole system under simulation (or internal events of simulated subsystems that have been explicitly marked to be traced).

Visualisation is useful in order to understand the system behaviour and system structure. Visualisation helps the designer to find the bottlenecks and errors in the system partitioning, thus enabling the designer to improve the simulation set-up's performance by re-partitioning.

The GA is connected to the model database and to the IOE. The model database access allows GA to show the names of the simulators as they appear in the modelling, while the IOE connection is used for animation itself. The GA receives the messages to be animated from the IOE. The IOE informs the GA about all of the sending and the receiving of the events in the MSE. However, if the designer wants to concentrate on certain signals, a filtering mechanism may be used for restricting unwanted events.

4.3 MSE SIMULATION PREPARING SEQUENCE

When the simulation is constructed, several operations must be done. First, the simulator's own model building i.e. creation of simulatable entities, design modules, has to be linked to the to the MSE. This may be a manual or automated procedure i.e. the simulator may have an automated tool which registers the piece of the design into the Model Database and writes the files required. The design module registration is not supported with a command line, since it is not a part of the system level model building, but rather a prerequisite for the system level model building.

After the design modules are registered on the Model Database, the system modelling can take place. The Graphical Model Builder (GMB) allows the integration of the simulator into the system modelling by executing a designer specified command line after the simulation model is constructed. The

command given in this phase is intended to allow the MSE integrated simulation environment to prepare a subsystem, a part of the system under simulation allocated to one simulator, for simulation. Typically this means that the automatic configuration generator is to be executed, but it may also be a text editor, or, if the simulator does not require any preparing, this command may be omitted totally.

The ready-made system model may be simulated by locking the simulation model to be simulated. After locking, the GMB executes start-commands for the simulation environment tools. Normally there is only one simulation environment tool, the IOE script generator, which reads the information of the simulation model and the simulator's start-commands from the MDB and writes them to a script-file. The start-commands are similarly designer specified, as was the case with the prepare-commands. Again the start-command may launch a fully automated simulation loader, or simply enable the designer to select and load a sub-design to the simulator. After the script is created, the script generator starts the IOE, which reads the start-commands of the simulators from the script file and executes them.

Figure 9 illustrates the MSE pre-simulation task as an eight phase workflow:

1. The Designer implements the design modules, for example, by writing VHDL code, and verifies their functionality by simulating. The Designer stores the design module in the MDB.

2. The Designer selects the design modules for the system model and connects the objects together with the SASD editor to form a system model.

3. The Designer selects a system model and creates a simulation model by allocating the objects to simulators, and connections to tracers.

4. GMB executes prepare-commands for each simulator in the simulation model. Each simulator prepares the simulation model for the simulation by making modifications and adding information to the simulation model.

5. The Designer locks the simulation model for simulation so that no other simulation model can be read accidentally from the MDB.

6. The IOE reads the locked simulation model.

7. The IOE executes start-commands for each simulator.

8. The Simulators establish the connections to the IOE and send an indication, that they are ready.

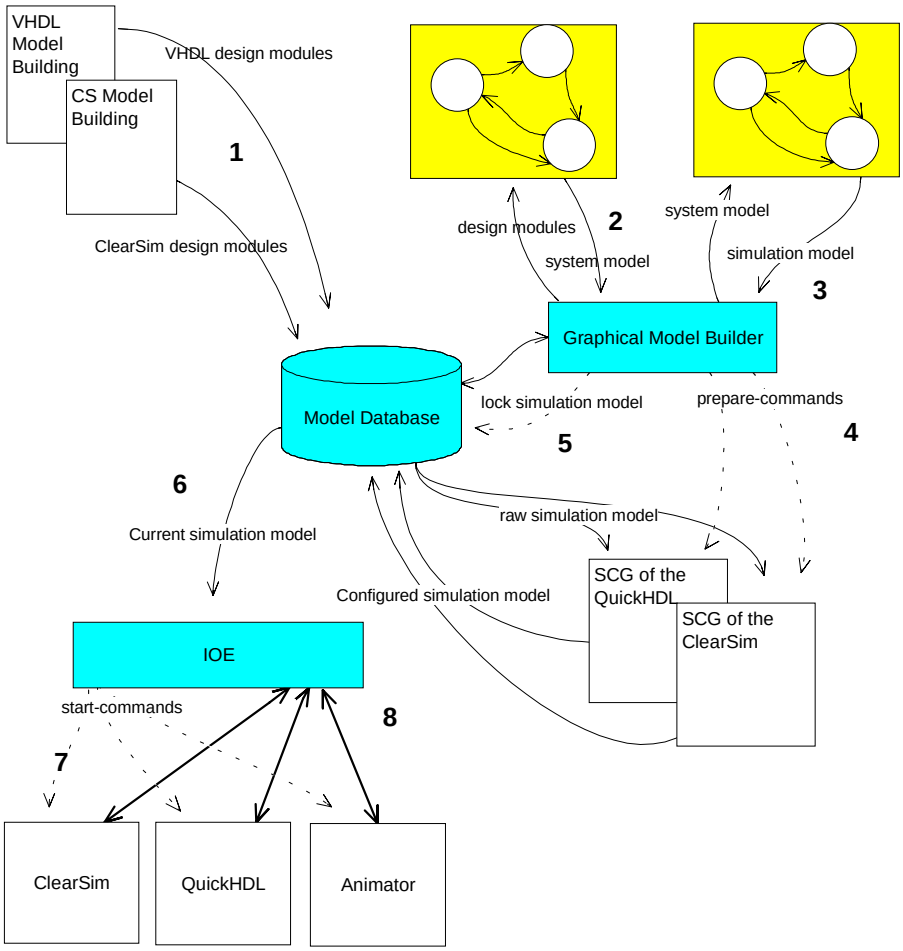


Figure 9. The usage of the MSE

4.4 MSE SIMULATION SEQUENCE

The MSE simulation sequence can be divided into three different phases: initialisation, session set-up and simulation phase. During the initialisation phase, the communication links are established between various tools in the MSE and the version of the communication protocol used is checked. The session set-up phase consists of information exchange of the system under simulation and the simulation set-up itself. Based on the information exchanged, the different simulators and tracers are able to communicate with each other. During the simulation phase, the simulators and the IOE communicate using the controlling messages for the simulators and the simulation events. The control messages are sent by the IOE to every simulator involved in the current simulation set-up, and their purpose is to query timing information and enable synchronous simulation of the system under simulation [29].

5. EMBEDDING VHDL SIMULATOR INTO THE MSE

5.1 OBJECTIVES

The objective in the embedding of the VHDL simulator into the MSE was to provide the designer with a hardware simulation capability. The methods developed are general and thus applicable to most VHDL simulators. In the OMI/TOOLS project, the methods were implemented and tested on a demonstrator with specific simulators. The VHDL simulating environment selected is QuickHDL from Mentor Graphics.

The MSE's basic idea is to enable the simulation and modelling of systems consisting of sub-designs intended to be simulated with different types of simulators. There are three basic requirements that the simulator must fulfill in order to be integrated into the MSE:

- The simulator must have the ability to understand the simulation model created by the Graphical Model Builder.
- The simulator must be able to respond to the external control events.
- The simulator must have the ability to exchange synchronisation and simulation data.

5.2 ROLE OF VHDL IN THE MSE

VHDL is one of the most commonly used hardware description languages around the world. In Europe, VHDL has almost totally displaced all other hardware description languages in higher level descriptions, although in gate level simulations the Verilog is also popular. Originally VHDL was intended for modelling, simulating and documenting electronic circuits. Almost every major Electronic Design Automation (EDA) vendor provides tools for VHDL simulation, directly or indirectly with cross-compilation. The ability to simulate

VHDL in MSE gives access to a vast amount of existing hardware models [6, 31].

VHDL's natural role in the MSE is to represent the application specific hardware parts of the system under development. It is also possible to use VHDL to describe standard, vendor provided processor cores and microcontrollers. For example, most microprocessor vendors provide VHDL models of their processors which can be used as a part of the system simulation. However, these are not the only possibilities, since VHDL supports the modelling of digital systems on several levels of abstraction ranging from the system level to the gate level.

5.3 FOREIGN LANGUAGE INTERFACE

The foreign language interfaces of VHDL simulation environments are used for connecting processes written in other languages than VHDL into the VHDL simulation. The connection is done by using the foreign attribute of VHDL. The VHDL standard version '93 states that the foreign attribute is declared in the package standard, in the earlier standard '87 the designer needed to declare the Foreign attribute himself [32]. Nowadays most workstation based VHDL simulators have the foreign language interface implemented as a C function library. The VHDL '93 standard does not specify the functions included in the interface. However, in practise the names and the parameters of the functions differ, but the services provided by the foreign language interface are similar.

5.4 IMPLEMENTATION OF THE INTERFACE

The VHDL side of the interface consists of entity and architecture description of a foreign language process. The entity description is similar to an ordinary VHDL process's entity. It describes the ports, defining the types, names and directions of the signals to be visible for the foreign language process. Only signals described in this entity can be driven or read from the foreign language process.

The architecture description contains an attribute clause, which tells the simulator kernel that the implementation is foreign to VHDL. The value of the foreign attribute is a string in three parts [32]:

- The first part is the name of the function to be executed first. This is usually an initialisation function, which allocates memory for signal drivers and signal ports.

- The second part is the list of files to load. The files are pre-compiled object files or libraries.

- The last part is a string passed to the initialisation function. This field is optional.

The foreign language part consists of two parts. The first part is the initialisation function mentioned above, and the second part is a simulated algorithm.

The initialisation function is called during the elaboration of the VHDL design. During elaboration, the simulator loads the processes from the libraries and links them together for the simulation. The initialisation function is called only once and it is used for allocating memory and for preparing the signal drivers for the simulation. Also the processes forming the simulated foreign algorithm and their executing conditions are presented in the initialisation function.

5.5 STRUCTURE OF EMBEDDED QUICKHDL

The aim in embedding QuickHDL in MSE is to create such an interface that the MSE can start the simulation without forcing the designer to control or configure the simulator. Additionally, the ability to compose different VHDL designs in a way that they can be simulated in one VHDL simulator was also desired. During the simulation, the QuickHDL was expected to send and receive simulation and synchronisation data and to respond to the control messages of the IOE. These requirements led to development of a separate model interface and Interoperation Engine Interface (IEI) for QuickHDL.

5.5.1 The Model Interface of the QuickHDL

The model interface of QuickHDL provides the interface between the Model DataBase (MDB) and the QuickHDL's VHDL compiler "qvhcom". The model interface consists of the Simulation Configuration Generator (SCG) and QuickHDL's own supporting tools and UNIX utilities.

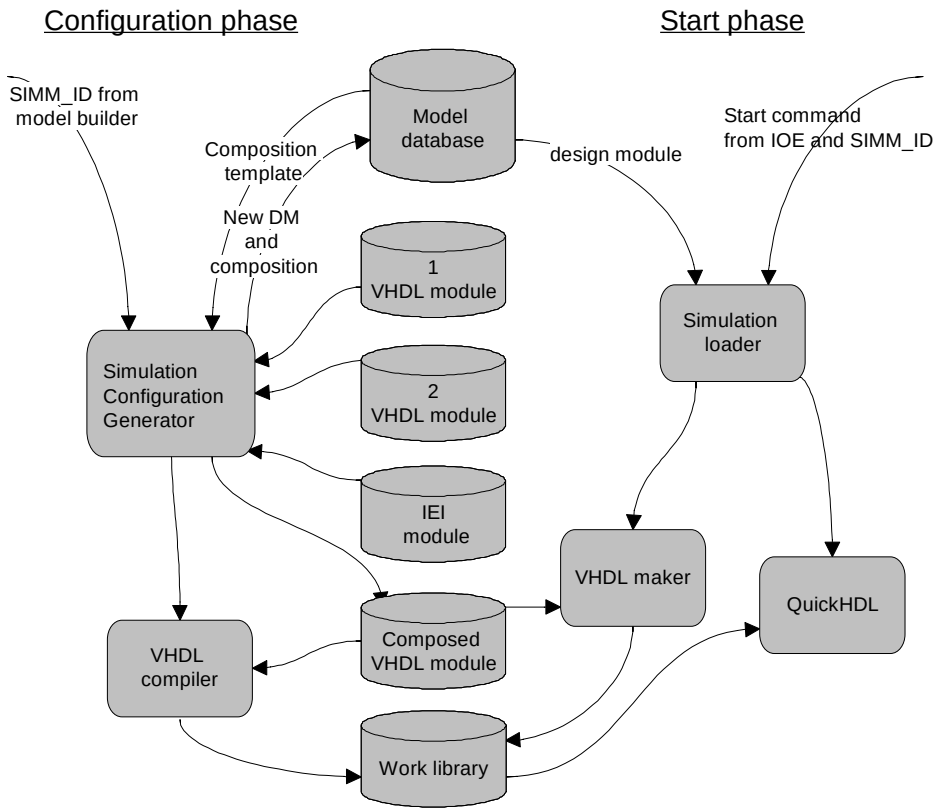


Figure 10. Functional view of the model interface of the VHDL simulator

In Figure 10 the whole design process from the model builder to the starting of the simulator is illustrated. The left hand side of Figure 10 describes the configuration phase. The configuration phase enables the QuickHDL's model interface to adapt the simulation partition to a form that can be simulated with QuickHDL simulator. The modifications needed are the creation of a top-level

entity and the insertion of the IEI to the VHDL description. The new top-level is needed so that the several design modules can be composed under single IEI, thus enabling the simulation of many design modules with one simulator. It is also convenient to reduce the communication since not all the intra design module communication does have to go trough the IOE, only the communication between the simulators is transferred trough the IOE.

The configuration phase starts, when the Graphical Model Builder (GMB) gives a start-command to the SCG. The start-command includes the information of the current simulation model id, SIMM_ID. The configuration generator loads the simulator configuration and the information of the related design modules from the Model DataBase.

The SCG composes all the design modules which are allocated to a single instance of the QuickHDL under one top-level entity. Even if there is only one design module allocated to one QuickHDL, the SCG creates a composition consisting of the IEI-module and one design module. The simulatable VHDL module is always a composition of at least two different modules, IEI and VHDL. A new design module is therefore needed to describe the composition. The design module created is similar to and has a similar syntax as the real design module. The major difference is that it can not be used for creating new compositions. This is due the attached IEI, since composing multiple compositions in one composition would also compose multiple IEIs in one composition and clearly that is not the thing to do. After the creation of the composition, the composition is verified by compiling with the same VHDL compiler as used by the target simulator.

It is possible to create a work library and makefile for the model. This enables the model to be simulated later without additional delay. The information of a possible makefile and the location of the work library is stored in a design module and the design module, is stored in the MDB.

The right hand side of Figure 10 describes the simulation start phase. The start phase begins when the IOE starts a Simulation Loader with a start command. The start command contains the information on the simulation model which enables the simulation loader to load the correct design module from the MDB.

If there is no makefile and work library described in the design module, a new work library is created. After the work library has been created, the simulation starter executes the VHDL compiler, which compiles the model into the work library. In case there is reference to a makefile in the design module, the UNIX make utility is used for verifying that the work library is up to date. The location of the source file for the compiler, as well as the name of the top-level module of the VHDL model, is got also from the design module. The top-level name is needed for starting the QuickHDL.

The IEI starts when QuickHDL elaborates the design tree. First the IEI executes an initialisation sequence in which memory is allocated and the signals are mapped to the simulator. After initialisation, control is given back to the simulator that completes the design tree elaboration. Then the IEI process is restarted and the simulator is now ready to simulate.

5.5.2 Simulation Configuration Generator

The task of the Simulation Configuration Generator (SCG) is to create a single simulatable VHDL module and to produce the information necessary for the Simulation Loader to compile and load the simulation.

These tasks can be divided into several sub-tasks:

- composing all the VHDL modules allocated to one VHDL simulator into one module. This is necessary even if there is only one VHDL module, since the IEI is handled as a module, too
- creating the configuration file describing the paths of the VHDL modules in the simulation
- creating the work library for the sources to be compiled, adding the path to the work library in to the configuration file
- mapping the directory of the work library as a work library for this compilation. This action is a QuickHDL specific part, and as a result a quickhdl.ini-file is created. The path for this file is also inserted into the configuration file

- compiling the VHDL sources into a work library. In this phase the simulator dependent compiler is used. All the models must be compiled into a work library before the composition file is compiled
- creating a makefile if the simulator supports it. Add the path to the makefile into the configuration file. The QuickHDL facilitates the “qhmake” command which creates automatically a makefile for the library
- adding the path of the composition file into the configuration file
- saving the information of the composition in the MDB.

The SCG is started by the Graphical Model Builder. First the SCG creates the directory for holding the files to be created in composing. The work library and the configuration files are stored in this directory. In order to compose the simulatable model, the SCG retrieves information on the models involved and their connections from the MDB.

The composition is created by generating VHDL code describing the connections between the design modules and IEI. After saving the composition as a VHDL source, the work library will be created. For the physical location of the library a directory is created. The directory is a subdirectory of the previously created design directory. The creation of the work library will be done by the “qhlib “command of the QuickHDL.

The work directory has to be also mapped as a library. This will be done with the “qvhmap” command of the QuickHDL. The “qvhmap” creates the logical mapping of the work directory to a work library. The information of the mapping is stored in the “quickhdl.ini”-file.

For verifying the success of the composition generation, the model is compiled. The compilation is done by compiling first all the involved VHDL modules into the work library, and then compiling the composition file. After the compilation of the model, the work library contains the model which is ready for simulation. For maintaining data consistency, the makefile is generated.

In Figure 11 the function of the SCG is depicted. On the left hand side is the Model DataBase (MDB). The information transferred between the MDB and the SCG are described as arrows. On the right hand side of Figure 11 are all the files created by the SCG.

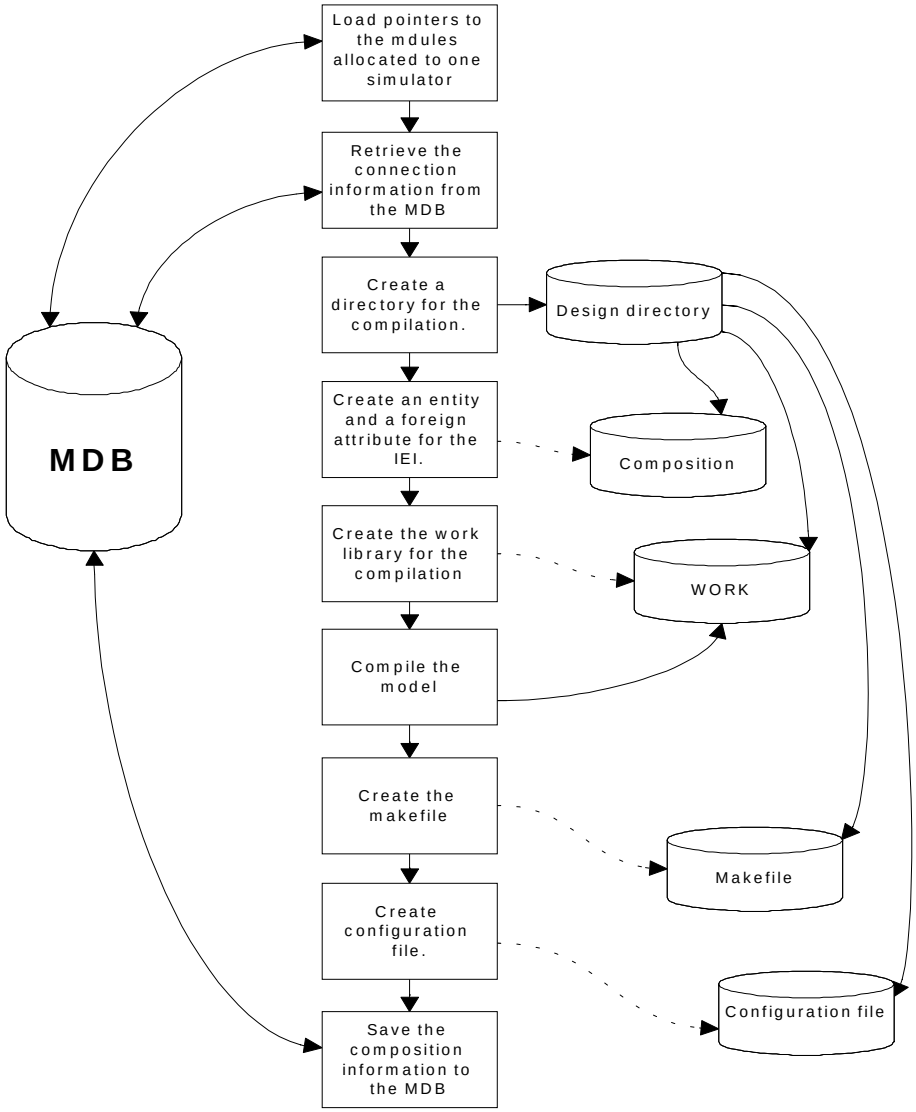


Figure 11. The function of the SCG.

For starting QuickHDL, at least the location of the work library and the top-level module in the composition is required. The top-level name is needed to start the design tree elaboration in the correct order. To ensure the successful elaboration, a simple configuration file is constructed. The configuration file contains the information on the source files locations and optionally the

reference to the work library. It contains also the reference to the makefile (if any), the location of the “quickhdl.ini” and the reference to the composition file. In other words, the configuration file alone contains the information needed by the simulation loader to start the simulation.

Composition file

The composition file is a file which contains the information on how to connect different VHDL modules to each other. The composition file is a VHDL code file which is an input file for the compiler. The compilation of the composition file is possible only after the VHDL-modules described in the composition file are compiled.

Let us consider a simple VHDL model which consists of two independent VHDL modules. In order to simulate these modules in one simulator connected in the MSE, the modules have first to be connected to each other and to the IEI module.

The result is a composition of two design modules and the IEI. Composition enables the more efficient use of simulator resources, since there is no need to add new simulator resources if a new design module is added to the simulation. Additionally, the composition reduces the communication between simulators. In Figure 12, the amount of signals transferred through the IOE was reduced from six to two.

Some VHDL code must be generated which describes all the connections between the modules. For the IEI, the connection information must also include all the signals which are marked as traced i.e. the signals that are internal to composition, but logically they are inter-module signals in the simulation model and thus possibly marked to be traced. The file created from the sample model of the previous page is shown in Appendix 1.

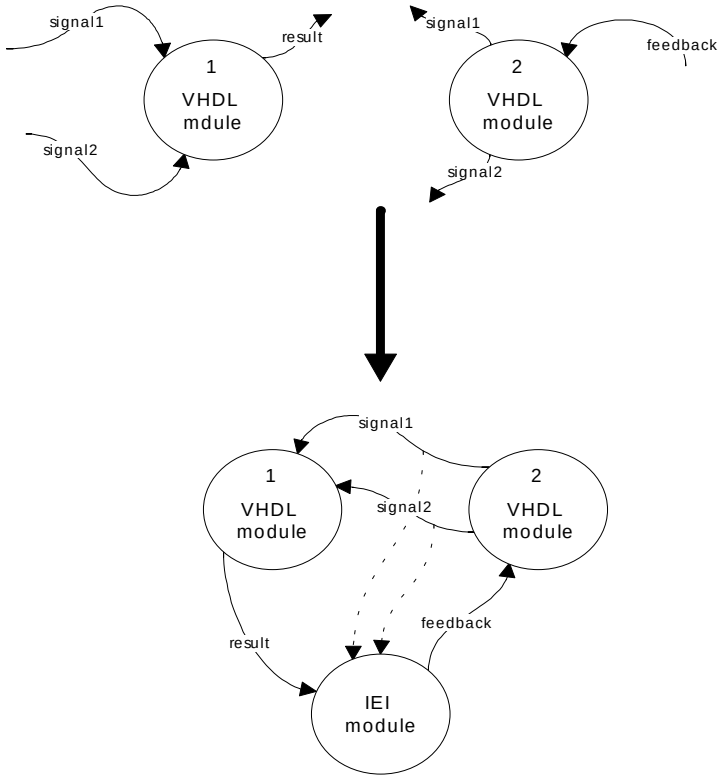


Figure 12. The Composition generation.

The VHDL file, describing the composition is created automatically. However, the creation of the component declaration requires the information on the entity declaration of the same component. Automatic generation of the interface from the VHDL code of the design modules would require full translation of the VHDL syntax, at least as far as entity declaration is concerned. The SCG takes a different approach; the signal order saved to the MDB is assumed to be correct. The order is defined by feeding the information to the MDB in the same order as in the entity declaration. The MIF API gives the ID numbers to the signals as they are fed to the MDB. The ID numbers are then used later to construct the correct order of those signals for the SCG. In practice, this means that the designer or the automatic model builder, is responsible for the signals being fed in the correct order into the MDB.

Configuration file

The configuration file includes information on the existence and locations of the files and libraries needed for verifying and starting the simulation. The information described in the configuration file is partially overlapping with the information drawn from the MDB. However, the pointers to a composition library and optional makefile are only described in the configuration file. A Single configuration file is convenient since the MDB may only save a pointer to one file instead of many.

The simulation loader uses to configuration file in order to find the top-level name of the VHDL design and the path to the work library. Based on these minimum requirements, the simulation loader is capable of starting QuickHDL. The name of the top-level of the model is given by the SCG. The SCG always creates top-level of the name COMBO, but the Simulation Loader can use other names as well. For safety reasons, the location of the makefile and the location of the quickhdl.ini file is also stored into the configuration file. This allows verification of that the simulated model is up-to-date and the simulator can find all the required libraries.

5.5.3 SCG's Interface to Model Data Base

Creation of Composition file

As the SCG starts, it retrieves the correct simulation model from the MDB. After retrieving the model, the SCG looks for the simulator resource ID it is supposed to create the composition. The simulator resource ID is located under the field:

GMB_SIMULATION_MODEL->SiM_Partioning->P_Simulator->SR_ID

The SCG browses the SiM_Partioning fields, until it finds the correct simulator. After finding the correct ID, the SCG calls MIF API function ERR_CreateComposition, which returns all the data needed to create the composition. There is no additional need for browsing the database further. The fields containing the objects to be composed (IDs), their original interfaces and all of their original input and output connections are described in Table 2.

Table 2. The data needed for composition creation.

GMB_COMPOSITION_INFO	contents of the filed
CO	Name-table (names and IDs) of the composition objects (objects to be composed).
PCOInfo	Structure of the composition objects' interface information.
Coinconn	Name-table of the connections to any of the composition objects.
Cooutconn	Name-table of the connections from any of the composition objects.
EO	Name-table of the ext. objects (objects NOT to be composed).
Eoinconn	Name-table of the connections to any of the external objects.
Eooutconn	Name-table of the connections from any of the external objects.
SiCObjectID	Composed object ID (0 if not created)
siNCObject	Non-composed object ID
Cinentityconn	Name-table of the connections to the composed object.
Coutentityconn	Name-table of the connections from the composed object.
Cintconn	Name-table of the composed object’s internal signals.

Libraries containing the design modules

The work libraries of the models to be composed are located at the beginning of the file. The paths for the libraries are got from the Design Module file. In order to enable the simulator to find these libraries, they have to be mapped into the local “quickhdl.ini”-file. After this, the information that is needed for compiling the model is written at the beginning of the file:

```
library module1
use module1.all
```

IEI’s entity declaration

Then the simulator creates the entity declaration for the IEI module. The information for this is found from the name-tables Cinentityconn and Coutentityconn. The SCG generates the IEI entity by first writing the entity declaration, then the connection is retrieved based on the id found from the name-tables. The name for the connection can be found from:

GMB_CONNECTION->C_Name

The name is written into the composition file together with the direction. The direction (in/out) can be resolved based on which name table Cinentityconn or Coutentityconn the connection id was originally found from. For the entity declaration, the type of connection is also needed. The type of connection can be found by selecting the composition side flow and checking the flow’s type. This means that the input connection type is the drain flow’s type and the output flow type is the source flow’s type. The drain flow type can be found from:

GMB_CONNECTION->C_Drain->D_DrainFlow->F_Type

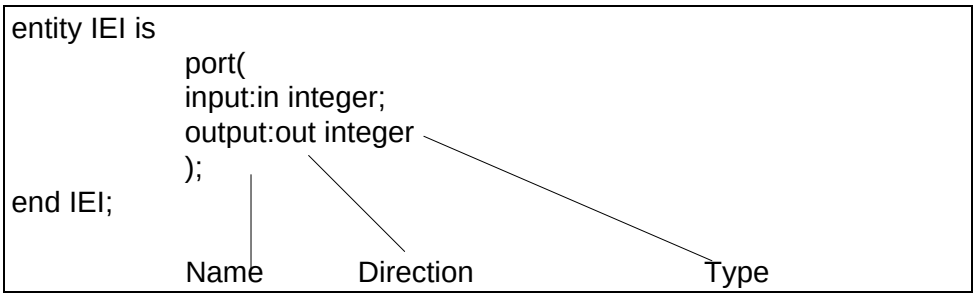
The one connection may have multiple drains, which means the SCG has to check all of these drains in order to make sure that the entity can be created.

The source flow type is easier to find, since one connection has only one source. The source flow type can be found from:

GMB_CONNECTION->C_SourceFlow->F_Type

The type of signal is an integer value which can be mapped into a name by function ERR_RetrieveFlowTypeDef. The function returns the structure, which includes the “real” name for the type of the flow.

The steps described are then repeated for all the connections in the Cinentityconn and Coutentityconn name-tables. After this the entity declaration for the IEI looks as follows:



The architecture of the IEI is described in the foreign language process. In order to find the correct file the foreign attribute is written into the composition file. The foreign attribute declaration does not need any information from the MDB.

Component declarations

The design modules involved in a composition are introduced in the component declarations. For the component declarations the information must be read from the MDB. The information for the composed object’s component declarations is found from the design module.

The information for the IEI component is basically the same as in the entity declaration. The component declaration for IEI can be created by using same routines as were used for the entity declaration.

The composition design modules can be found by first browsing the objects to be composed. The creation process itself is similar with entity declaration, but the information needed is located differently The name-table of the objects involved can be found from:

GMB_COMPOSITION_INFO->CO

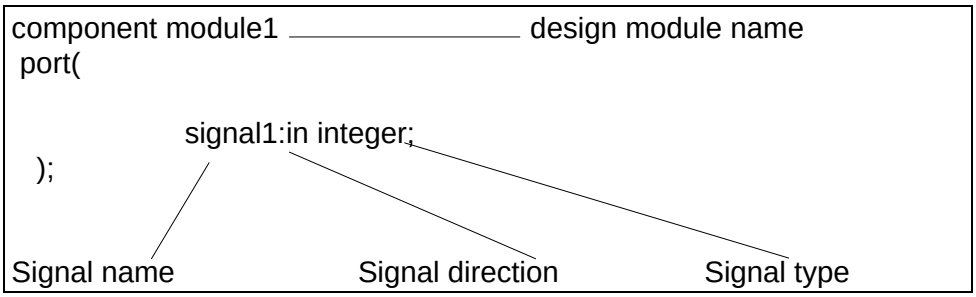
The name-table includes the id numbers of the objects. Based on the id numbers, the objects can be retrieved from the MDB. The object includes the design module so the name of the design module can be found directly from:

GMB_OBJECT-> O_DesignModule->DM_Name

The names of the signals can be found under:

O_DesignModule->DM_Flow->F_Name

The types of signals are checked so that all the ends (drains and sources) are of the same type.



Signal declaration

For the signal declaration, the name-table of the connections is found from the COinconn and COoutconn. All the connections under these fields are declared in the signal declaration.

The name-table of the connections includes the id number of the connection. Based on this information, the connection can be retrieved from the MDB. For the signal declaration the name can be found from:

GMB_CONNECTION->C_Name

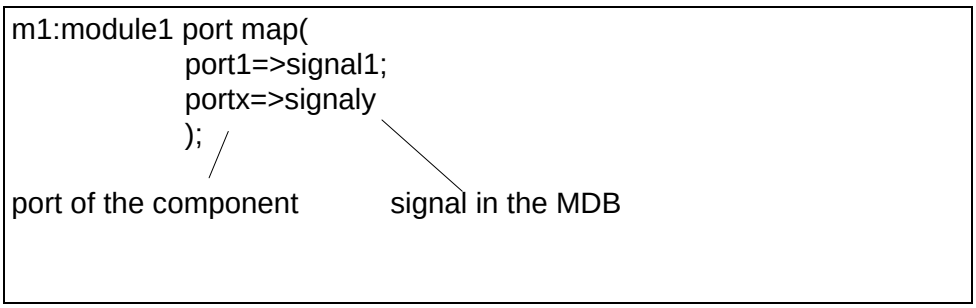
The signal type can be found by checking the source of the signal, if the source is not part of the composition, then we must use a drain of the signal. Since it is possible that there are numerous drain modules for a signal, their datatypes must be equal in order for the composition to succeed. The method is similar to that in the component declaration.

Connecting the components

The connections between modules(and between modules and IEI) are described in a portmap. In the portmap, the name of the port is attached to one of the signals of a signal declaration. For the IEI, the portmap includes all the external signals mapped to the ports. Since the name of the port in the IEI was the same as the connection names in the Cinentityconn and Coutentityconn.

For the other modules, the information on the port connections must be browsed by one object at time.

After all the signals have been checked for the module, the portmap looks like following:



Creating a new Design Module for composition

When the composition is created, the location of the composition design module file is stored in the MDB. This is done with the function: ERR_StoreAsDesignModule. The function returns a design module Id, which should be saved for later use. In order to later connect the design module to the other design modules on the system level, the ports of the module should also be

defined. The ports are added with the command: ERR_StoreAsFlow. This function needs the design module id, port name, port type and the port direction as input. The function returns the id of the flow. The StoreAsFlow function stores only one port at a time, so that it must be used once for every port. The type of flow needs some work, if the signal is an output signal from the composition, the type is the same as the corresponding signal in VHDL module. If the signal is an input signal, the type must be defined by looking for the original drains in the VHDL modules.

After the ports of the new design module are defined, the original modules must be replaced by composition in the simulation model. This can be done, by using the function: ERR_AddDesignModuleToComposition. As an input, this function needs the id of the simulation model, an id of the simulation resource(simulator) and the id of the design module. All the information needed should be directly available in this phase.

Now the design module is ready, and the original modules are replaced by the composition; however, the composition should still be connected to the rest of the design modules. This procedure can be done by using the ERR_BindCompositionConnection function. The function needs the simulation model id, resource id, port id and connection id. This function connects one port at a time, so it must be executed once for every port.

5.5.4 Interoperation Engine Interface

The task of the Interoperation Engine Interface (IEI) is to enable the IOE to control a simulator. The IEI provides the means to start the simulator easily with one start command. It establishes the connection to the InterOperation Engine (IOE) and reads information on the connections between the QuickHDL simulator and other simulators involved in the set-up. The IEI tries to automatically convert data types used by the IOE into QuickHDL format. During the simulation, it enables QuickHDL to send and receive signals to/from other simulators; it also enables the IOE to synchronise the simulation by allowing the simulation to run in small steps. The IEI delivers the information on the current state of the simulator to the IOE so that the IOE can calculate the next time step and drive the simulators in the correct order.

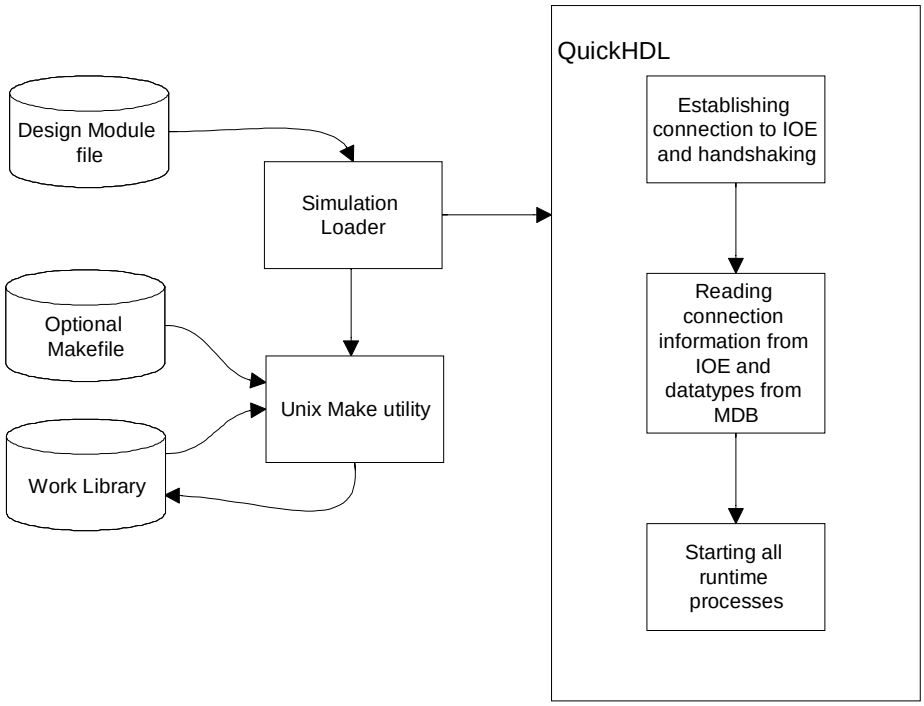


Figure 13. Starting the simulator with the IEI.

The IEI consists of two parts: the Simulation Loader that loads the simulation and a set of processes which controls the simulator.

The IOE starts the Simulation Loader with a start command. The start command contains the location of the design module file (Composition) to be simulated. The Simulation Loader analyses the contents of the design module file. During this analysis it checks the existence of the VHDL design files described in the design module and the library containing the simulatable VHDL design. The Simulation Loader can use an optional makefile to ensure, that the library is up-to-date. If the makefile exists, the UNIX Make is executed with this file. If the files and directories pass the check, i.e., the library described in the design module file exists, the simulator is then started with the name of the top-level entity described in the configuration file. The Simulation Loader consists of two parts. The first part is a starting script which is used to set the MGC_HOME and the license server environment variables. After setting the variables, the script starts the executable part. Both the script and the executable part take the same

command line variable, path and name of the design module. The script is named QuickLoad and the executable QL. The procedure for starting the simulator is described on the left side of Figure 13.

After the Simulation Loader has been executed, the simulator starts and elaborates the simulation object described in the library. The elaboration of the simulation object triggers the initialisation process. First this process establishes the connection to the IOE and then executes the initialisation phase. The initialisation phase is described in detail in the InterOperation Engine Communication Transmission Protocol (ICTP) [29] document. During the initialisation phase, the IEI receives the information on the external connections of the simulator. Based on this information, the IEI prepares itself to handle all the external signals. The simulator starting sequence is depicted on the right side of Figure 13.

When the IEI has initialised itself, it gives the QuickHDL kernel an order to execute the runtime process of the IEI. The runtime process handles the ICTP message receiving and the signal driving during the simulation phase.

5.5.5 Simulation Loader

The Modeling and Simulation Environment (MSE) of the OMI/TOOLS project demonstration contains two simulators, ClearSim and QuickHDL. However, the environment is intended to be open to integration of other simulators as well. This approach reduces the possibility of the environment controlling the simulator, since it cannot feature simulation depended control mechanisms. Yet still it should be possible to automatically configure the simulator to simulate the allocated design modules and to start the simulator without the need for user interactions. In order to manage these tasks, the simulator interfaces with the Simulation Configuration Generator (SCG) to MSE modelling and with the Simulation Loader to the MSE simulation.

The MSE allows the integrated simulation environment to use design module files for defining information needed to start the simulation. The contents of these files are meaningless to all the other tools connected to the MSE, other than the SCG and the Simulation Loader of the target simulator. The tasks of the

SCG are checking, compiling, combining or even creating these design module files, by interacting with the Model DataBase and the Graphical Model Builder.

The Simulation Loader interprets the design module file, checks the information for validity and starts the simulator, with following command:

qhsim <toplevel>

where the toplevel is the name of the top-level entity.

5.5.6 Foreign language process

The QuickHDL FLI has no direct function for stopping or pausing the simulation, but this facility is not needed since the execution of the FLI process blocks all the other QuickHDL processes. Thus, blocking can be easily used for stopping the simulation.

There are also some drawbacks in using an FLI process as a control interface to the simulator. Since the FLI process blocks all the other processes, it does so even to the user interface process and reduces the user's ability to affect the simulation. This reduced level of interactivity might not be a problem, considering the fact that it is a complete system which is simulated and there are several simulators involved.

The FLI part of the IEI is executed in two phases: the initialisation and runtime phase. The initialisation phase adapts the IEI for the selected simulation and the runtime phase handles the simulation control.

The IEI is linked to the design as a pre-compiled object. Also the individual VHDL modules are in the same format. When the simulator starts, it elaborates the design tree i.e. browses and links these objects to the simulation kernel. If the foreign attribute is defined in a VHDL module, the QuickHDL tries to call the function described by the attribute during the elaboration.

The FLI part of the IEI is defined as a foreign attribute on the top-level of the VHDL design. When the elaboration starts, the QuickHDL calls the FLI part of the IEI and the initialisation begins. The function call structure of the

initialisation phase is described in Figure 14. One function Initial_error_handler is not connected to any other function, but actually it is possible to be called from all the initialisation functions.

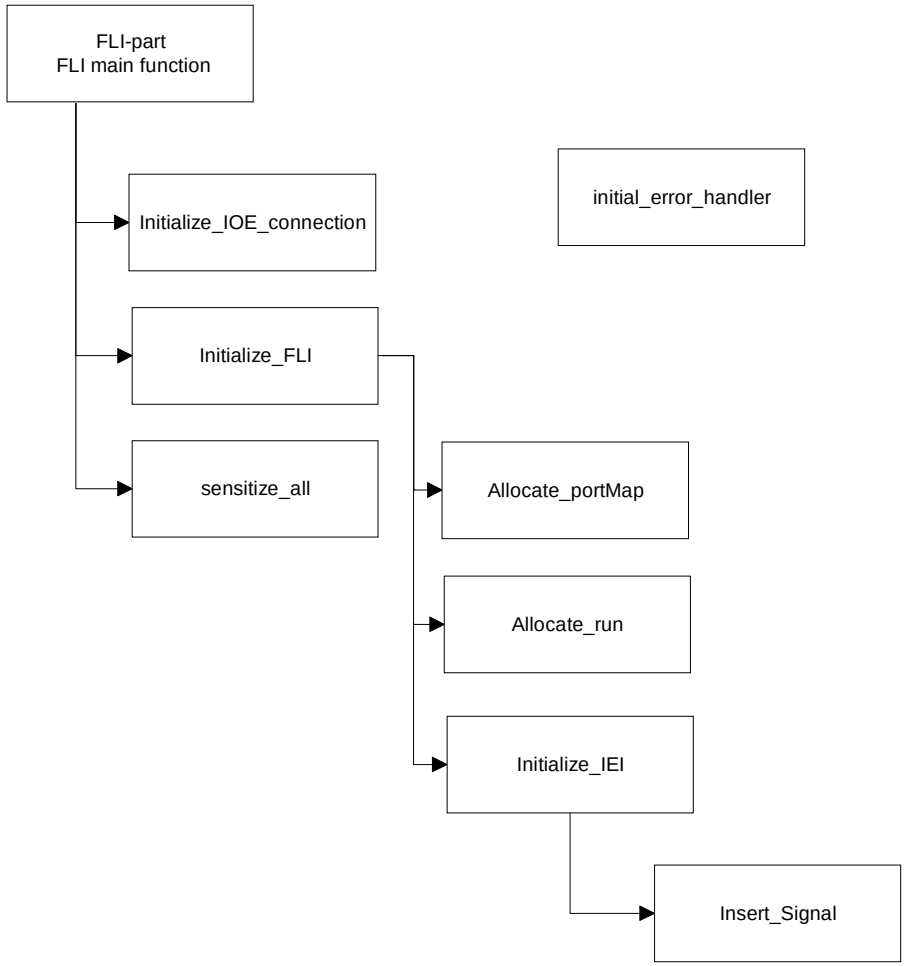


Figure 14. The function call structure of the IEI.

The FLIpart is the main function of the IEI. It calls other functions during initialisation and, in the end, sets the starting time of the runtime functionality.

When the IEI is started, the Initialize_IOE_connection -function is called; it establishes the socket connection and agrees on the protocol with the IOE. The detailed description of this phase is described in the document InterOperation

Engine Communication Transmission Protocol, section 5.3 [30]. The IEI allows the messages described in Table 3 to be transmitted during the initialisation. The usage of the other ICTP-messages causes the ICTP_NOTOK or ICTP_ABORT messages to be sent, and finally the termination of the connection. ICTP_ABORT is not normally used, it is reserved only for error situations.

Table 3. Messages allowed during handshaking.

Received message	Answer
ICTP_SERVER_HELLO	ICTP_CLIENT_HELLO
ICTP_VERSION	ICTP_OK / ICTP_NOTOK
ICTP_IOE_INI_ACC	ICTP_IEI_INI_ACC

After the connection is established, the Initialize_FLI function is called. This function allocates the dynamic memory structures used by the IEI and calls the Initialize_IEI function. The Initialize_IEI function handles the signal and driver mapping between the external and VHDL signals. It receives the information on the signals from the IOE and creates ports for the output signals and drivers for the input signals. The information on the signal-driver and signal-port pairs is stored in the dynamic memory structures.

The IEI creates drivers for the input signals which handle the writing of the signal transactions at the correct moment of time. A signal driver is a service provided by the QuickHDL. A signal driver can be used just by giving it a value and a time of occurrence in the simulated model. Even multiple successive values of the signal can be in the queue of one driver.

The output signals need a special mechanism since they can be used also to change the current running of the simulation, for example, from “running” to “wait for command”. The output signals of the simulated model are handled by output_processes. These processes share the same code but have different functionality depending on the type of the signal to which they are attached. The output processes are created in the initialisation phase and each process is attached to one signal. In this way one can sensitise the processes to signals. The

sensitising is done by telling the simulator kernel that a certain function, output_pro, must be called with a parameter, signal_id, in case there is a transaction in the triggering signal. The sensitising of the output signals is done in the function sensitize_all.

The messages used in the initialisation phase are described in the document InterOperation Engine Communication Transmission Protocol [29]. In the initialisation phase the IEI of the QuickHDL does not need to respond to all the messages sent by the IOE. The messages vital to the IEI are described in Table 4. Additionally, the IEI may send or receive an ICTP_ABORT message if an error occurs. Although the IEI ignores all the other signals, the IEI has the ability to recognise all the signals used in the Initialisation phase, so that any unknown message results in an error.

Table 4.Vital messages for Initialising the IEI.

Received Message	Send Message
	ICTP_TOOL_INFO
ICTP_CUT_SIGNAL_INFO	
ICTP_IOE_SSUP_ACC	ICTP_IEI_SSUP_ACC

Simulation time functionality

Figure 15 illustrates the messages passed between the IEI and IOE during a typical simulation. The lighter coloured boxes describe the messages sent by the IEI and the darker boxes messages sent by the IOE.

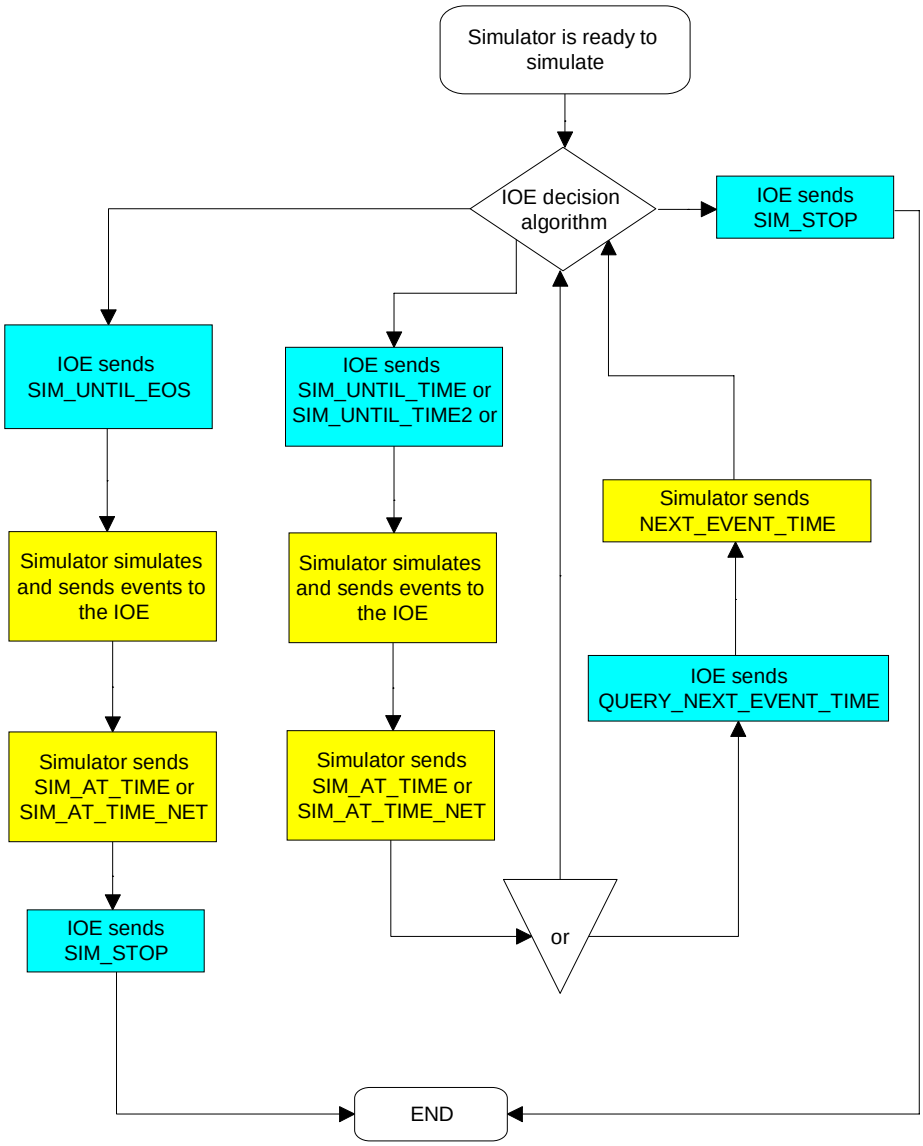


Figure 15. Typical simulation controlled by the IOE.

After the elaboration of the design tree, the Runtime_Master function is called. This is done by setting the Runtime_Master function as an ‘elaborate done’ callback function. This means that the Runtime_Master function is the first function called by the simulator kernel after the elaboration phase has been completed. The Runtime_Master is a process that handles most of the ICTP-

messages during the simulation run. The Runtime_Master's task is twofold: it handles the run commands from the IOE and the incoming signals from the other simulators. However, the Runtime_Master does not handle the output signals and end-of-simulation situations.

The control messages accepted by the IEI consist mainly of run commands and simulation time queries. The Runtime_Master make the simulation progress in several ways. The simplest way is to run it until the end of the simulation has reached. This is done if the IEI receives a SIM_UNTIL_EOS message. In this case, the Runtime_Master is run only once, after which control is given to the simulation kernel which runs the simulated model to the end. This means that all the incoming messages, as well as any control messages from the IOE are omitted. However, all the output signals of the simulated model are written to the IOE. After the simulation has reached its end, the simulator kernel calls the End_Of_Simulation function to inform the IOE that the simulation has reached its end and that the connection may terminate.

The simulator can be set to run until a selected simulation time is reached. SIM_UNTIL_TIME message orders the IEI to wake up the Runtime_Master after a time step has elapsed. The time step is given with the SIM_UNTIL_TIME message as an absolute simulation time, not as a delay.

The most versatile run command is SIM_UNTIL_TIME2 message. The SIM_UNTIL_TIME2 message contains two time-stamps, the first time-stamp is the value for SIM_UNTIL_TIME message. After the time is reached the messages are not read from the IOE but the simulation continues until the event occurs or the second time-stamp is reached.

When the Runtime_Master starts, it normally sends a status message to the IOE which includes the current time of the simulated model and the time of the next event in the model. After this, the Runtime_Master waits for messages from the IOE. Only the SIM_UNTIL_TIME2 message omits this by continuing the simulation until second time-stamp, or until there is a event.

When the Runtime_Master is run for the first time, it creates two processes: the End_Of_Simulation process and another instance of itself. The End_Of_Simulation process is activated whenever the simulation changes its

status, for example from “running” to “idle”. The End_Of_Simulation is also called if the Runtime_Master is called several consecutive times without executing any VHDL process between the calls. This usually means that there are no VHDL events to come any more but only the Runtime_Master itself. The End_Of_Simulation process decides if the simulation has reached its end and if so, terminates the IEI. The End_Of_Simulation function informs also the IOE that the simulation has reached its end, and no more run commands can be accepted.

The outcoming signals are handled by the output_pro process. When wakened, the output_pro process decides which type of event it should create for the signal. Then it attaches the time of the occurrence and the value of the transaction to the event and sends the event to the IOE. If the previous command was SIM_UNTIL_TIME2 and the first time stamp was already reached, the output_pro returns the control to the Runtime_Master, otherwise the control is returned to the simulation kernel which continues the simulation.

The runtime structure of the IEI is described in Figure 16.

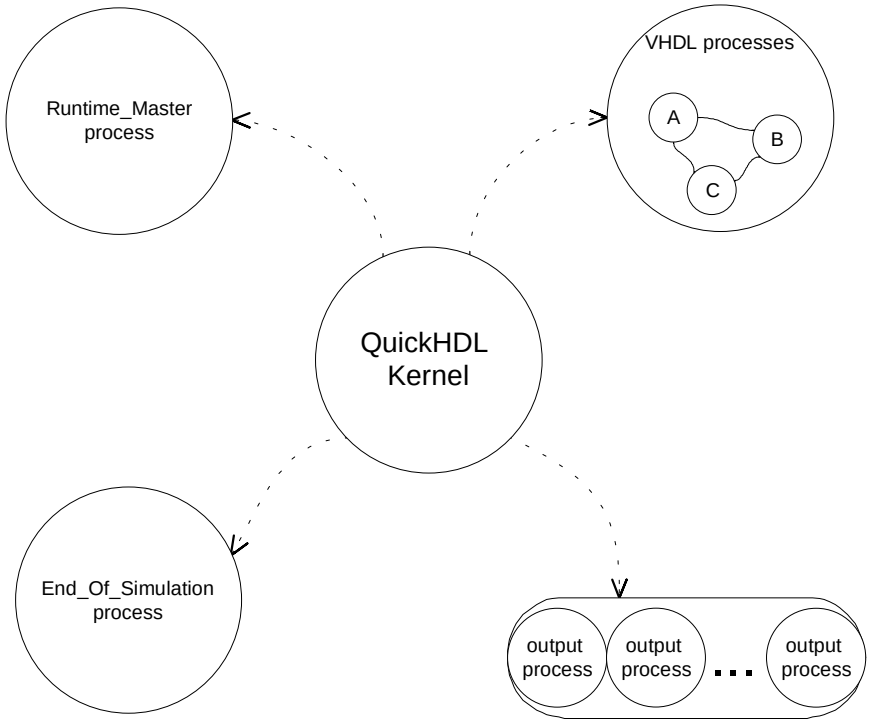


Figure 16. Runtime structure of the IEI.

Libraries used by the IEI

The IEI utilises several types of function libraries used in many different phases. A GNU Multiple Precision Arithmetic Library [33] is used for conversion routines which requires calculations using 64 bit long integers. QuickHDL Foreign Language functions [32] are used for all the interactions between the simulated model and the IEI. This includes also the scheduling and controlling of the simulator. The IOE/IEI Communication Library [30] functions are used for handling the data exchange between the IEI and IOE.

6. EVALUATION OF THE CO-SIMULATION IN THE MSE

The evaluation of embedding the QuickHDL VHDL-simulator into the MSE is done by constructing a simulation model with the Graphical ModelBuilder (GMB) and by co-simulating the resulting model with the ClearSim2 and VHDL simulator interlinked by the IOE. The performance of the co-simulation environment is then evaluated as a whole rather than just from the embedded QuickHDL's point of view.

The evaluation of the co-simulation environment suffers from the absence of a common scale that would enable real comparative analyses with existing methods. The measurement difficulty is caused by the dependency of the performance on the model simulated and the measured parameter. For example, the accuracy of software simulation is hard to detect since it depends largely on the application and the processor type. Pure accuracy of the system simulation can be measured only by comparing the simulation results with the execution of the real prototype. This is not, however, possible during this project due to the lack of a physical prototype, thus limiting the accuracy of the results only to estimations.

In the performance study, the situation is better, since the actual time consumed to execute a simulation can be measured and the effect of the communications on the speed calculated. In this work, the actual execution time of the simulation is called the simulation time, and the time advanced by the model, the model time.

6.1 MEASUREMENT ARRANGEMENTS

In order to test and measure the speed and usability of the simulation environment, test models are required. Ideally the model used for the performance studies should be as close to a real application as possible. However, it is not possible to define a single, right, real application. The OMI/TOOLS environment is intended for the co-simulation of complex embedded systems and especially for system-on-a-chip circuits. Still, the target

environment may vary a lot, for example, a 10 000 gate system-chip with a simple 8 bit micro controller core can be as real example as a 1 000000 gate system with a state-of-the-art processor core and multiple DSPs. However, during the simulation it does make a difference which model is simulated, since it is faster to simulate a simple model than a complex one.

The figure which is usually mentioned when speaking of the performance of the co-simulation environments is the instructions per second. It gives a rough estimate of how many assembly instructions are executed on average during one second of simulation time. Similarly the execution speed of the hardware simulation could be measured, but since there is no such a figure as hardware instructions per second, the benchmark is not useful.

The instruction per second figure is strongly dependent on the simulation models' interaction rate between the software and the hardware. The reason for this is that the simulators are much faster than the communication channel between them. Also the amount of software compared to hardware in the simulated model has its effects on the simulation speed. In general, the software simulation is faster than the hardware simulation, which tends to make the simulation of custom hardware dominated systems slower than the simulation of software running on a standard core processor.

For the above reasons, the selection and construction of the test models must be careful in order to achieve results that are comparable with the figures given to existing solutions.

The real co-simulation performance is measured by connecting QuickHDL and ClearSim2 simulators via IOE together and executing the simulation. The execution time of the simulation is measured and the test is repeated with several different rates of communications between two simulators. This data gives the estimates of the effect of the communication into the simulation performance. Tests are first executed when all simulators are in the same workstation and then in a distributed manner, where several workstations are used over the network.

6.1.1 Environment

The environment, i.e. the workstations, operating systems and network have their own effect on the simulation speed. It is understandable that a new fast workstation will outperform a model couple of years old. Still the co-simulation has a tendency to be restricted also for other reasons than the pure CPU speed, such as communication and synchronisation between simulators. From the communication point of view, the network and operating system play the most important role. The co-simulation backplane IOE has the greatest effect on synchronisation, which requires that the IOE's host workstation must be as powerful as possible.

For the test simulations, the network used is the 100 Mbit Ethernet, which is a quite typical solution for the modern workstation based network. The faster of the two workstations used is a SUN SPARC ULTRA 1 Creator with a 167 MHz processor, which is a two year old workstation model. Another workstation used is the SUN SPARC ULTRA 1 with a 143 MHz processor. Both workstations are equipped with the SOLARIS 2.5 operating system, which offers effective access to the network. Although these workstations do not offer the state-of-the-art performance, they still represent the common workstation configuration found in industry today.

6.1.2 Models used

There are two possibilities for constructing a test model. One possibility is the use of a complex system design. The drawback is that the performance figures measured would be typical for the tested model and they might be totally different with some other simulation model. Another possibility is to construct a suitable model which enables more generic results to be measured.

There are no common testing models that are used by EDA vendors to measure the performance of co-simulation environments. However, some figures are available, e.g. Mentor Graphics announced in the EE-Times article [34] that the typical amount of instructions executed after every I/O cycle is between 200 to 1000. This seems to be valid figure for a small system, but for a more complex system the figure can be much higher.

The test model for the MSE consists of software executed in a software simulator which models the execution of software on a 16 bit microcontroller, and of VHDL description simulated in the VHDL simulator. The VHDL description used is a fast behavioural level description in order to limit the effect of the performance of the hardware simulator on overall performance. The simulation of a complex VHDL model would only measure the performance of the VHDL simulator, but not that of the co-simulation. The rate between I/O operations and normal instructions is made easily alterable in order to allow the testing of the effect of the I/O on the simulation performance.

Fibonacci test-model

The Fibonacci sequence is a very famous sequence of numbers. In a Fibonacci sequence, every member is the sum of the two previous members. The Fibonacci sequence starts from two 1's and then proceeds as follows:

1, 1, 1+1=2, 1+2=3, 2+3=5, ..

Based on the Fibonacci sequence, it is possible to create a test model which features a simple structure and provides a heavy load on one simulator. The idea of the Fibonacci test is not to demonstrate the partitioning of the calculation algorithm between two simulators, but just to cause enough workload. The Fibonacci calculation is partitioned to be handled by ClearSim2 and the control of the calculated loops is partitioned to the task of the VHDL simulator.

The advantage of the Fibonacci model is that the load caused to simulators can be altered by one parameter, the degree of the Fibonacci calculated. The model can be set to cause only a very slight load on software simulator, thus enabling the communication i.e. the co-simulation performance to be tested. The same model can also to be set to give very heavy load on the software simulator, thus reducing the communication rate between simulators. This method enables the effect of a more complex model to be measured.

Heater Controller model

The model described in the previous chapter enables easily interpretable results to be measured, but the model itself is far from a real application. In order to

overcome this drawback another model is required to ensure that the results got from the Fibonacci test are correct. As a reference model a small control application, heater controller, is used. In the heater control model, the load caused to the simulator is not so easily alterable as it is in the case of the Fibonacci test, but the algorithms executed by the simulators are more complex, thus enabling more practical simulation results.

In the heater controller there are two design modules, the controller and the heater. The heater module contains the model of a heater, an environment and a temperature measurement unit. The controller tries to achieve a pre-defined temperature and maintain it by controlling the heater. Two design modules are connected with three signals: full_power, half_power and temperature.

6.2 PERFORMANCE MEASUREMENTS

The simulation results of the Fibonacci simulation model executed in one workstation are illustrated in Figure 17. The graph is based on the actual simulation results presented in Appendix B, Table B - 1. The figure depicts the relationship between the performance and the amount of machine instructions executed before an I/O event occurs. It can be seen that the achieved co-simulation performance is clearly dependent on the ratio between the I/O-events and the processor instructions executed by the software simulator. However, after a certain point, the speed of the software simulation becomes a bottleneck itself. After that point, it is not possible to improve the performance by reducing I/O events. Unfortunately, in practice many of the co-simulation problems require a simulation speed at the lower end of the curve, around 500 to 5000 instructions before an I/O-event. At the I/O ratio of 5000 instruction, about half of the maximum speed is achievable. At the lower end, the excessive communication causes the network to be saturated, which reduces the achievable co-simulation speed. It is also likely that a more complex VHDL-model would reduce the performance at the lower end of the curve.

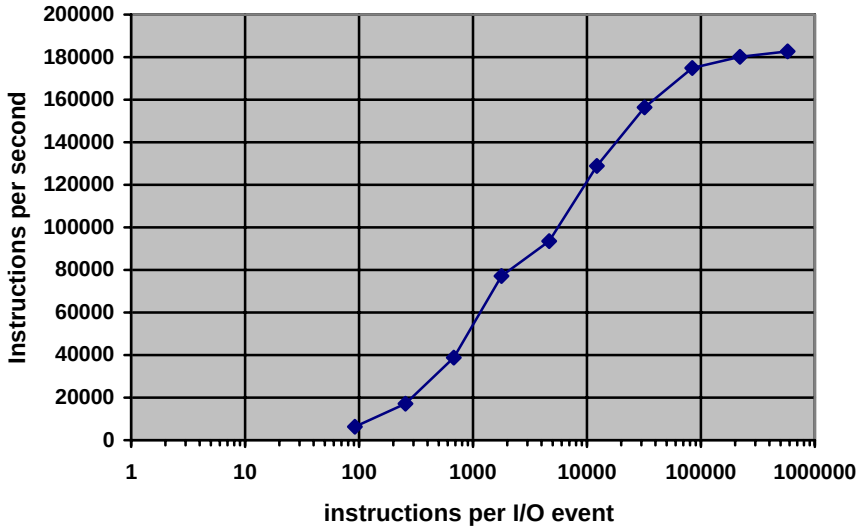


Figure 17. Performance of the MSE measured with the Fibonacci test and single workstation.

Figure 18 illustrates the performance of the MSE in the simulation of the Fibonacci test model distributed between two workstations. The actual values for the curve can be found from Appendix B, Table B - 2. Figure 18 reveals that with the simple model the performance of the distributed simulation is actually worse than it was with only one workstation in use. The main reason for this is that the host executing the software part of the simulation is a slower workstation than was the case in Figure 17.

In Figure 19, the situation is corrected by repartitioning the QuickHDL and IOE to the slower workstation and executing the software part in the faster workstation. Even after this modification, the final performance is only approximately the same as was the case with only one workstation. It is, however, remarkable that the simulation is not slower either, since this allows to distributing the simulation if required for other reasons than pure performance. For example, this may be needed for simulators requiring different operating systems. It is also likely that larger simulation models benefit more from the repartitioning and distributing to several workstations, since they tend to consume more workstation resources e.g. disk and memory. Also the large models, without global feedback, i.e. no cycle-wait type behaviour, can benefit

from the distributing. The actual values for the curve can be found from Appendix B, Table B - 3.

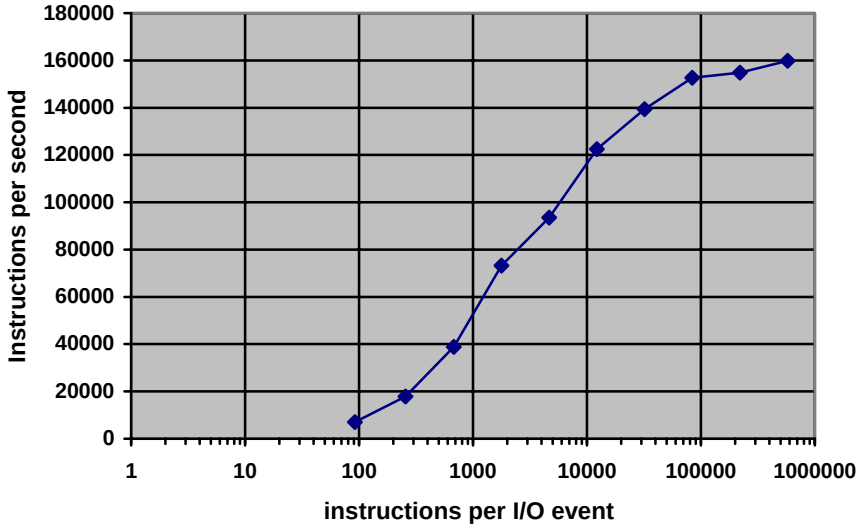


Figure 18. Performance of the MSE measured with the Fibonacci test and two workstations. The software is simulated in the slower workstation.

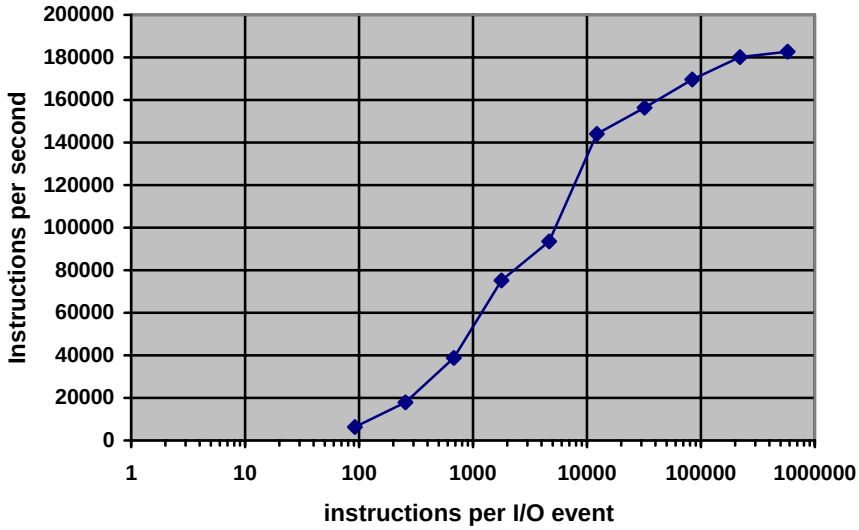


Figure 19. Performance of the MSE measured with the Fibonacci test and two workstations. The software is simulated in the faster workstation.

Figure 20 reveals the performance bottleneck of the synchronous co-simulation. If the IOE cannot determine the difference between a potential external event and an internal event, it runs the simulation in small time steps that make the simulation ineffective. The inability to determine whether the event should be considered as a potential external event is not actually the IOE's problem, but rather the simulator's problem, since the IOE makes the decisions based on the information from the simulator. However, the simulator's modelling technology, as is the case with VHDL, causes all the events inside the VHDL model to be considered as possible external, I/O-events. This should be considered when the model is written since there is not much to be done when the model is simulated. The heater model contains a lot of internal events, which are declared to happen after certain time delays. This approach causes the IOE to simulate these delays one by one in order to make sure that the possible external events are handled and reacted to in correct time. In most cases this is unnecessary since there is only one signal in the heater model that can cause I/O-event.

The curve in Figure 20 is drawn by keeping the value of processor instructions executed by the software simulator per one I/O-event constantly at 47, while the frequency of the samples taken from simulation is varied. It can be seen that when the sampling rate is kept high, i.e. almost all events that occur inside the VHDL-simulation are inter-simulator events, the instructions per second value correlate well to values measured in the Fibonacci test. However, when the sampling rate is lowered the performance of the simulation environment is radically worsened. This is strictly due the excessive communication caused by the synchronisation. The amount of actual communication is at the same level, or even less, but this time they can be considered unnecessary since the synchronisation events do not contain any simulation data. The actual values measured for the heater model can be found from the Appendix B, Table B - 4.

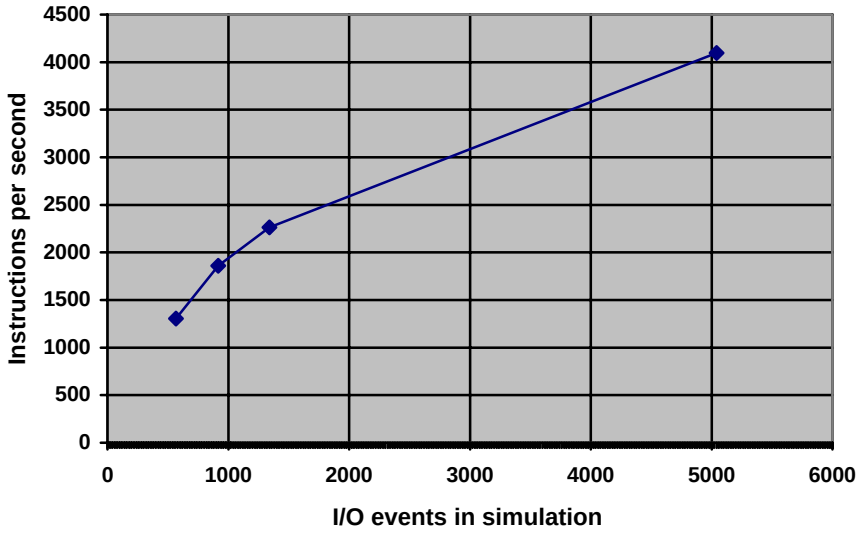


Figure 20. The speed of the heater simulation, with four different sampling rates.

6.3 COMPARISON TO COMMERCIAL SYSTEMS

There are currently two co-simulation environments on the market which are considered to be aimed at the same markets as the OMI/TOOLS MSE. These two are the ViewLogic EagleI and Mentor Graphics' Seamless CVE. These two commercial products have a lot of common features. They are both targeted for general applications, allowing the co-simulation of the HDL and the software written for the target processor that is presented as a processor model in the system. They are based on connecting existing simulators with a simulation backplane, i.e. they are heterogeneous co-simulation environments. The execution of the software is viewed in the HW simulator side as a processor bus model. This means that the target processors interface is modelled inside the HW simulation, but the actual functionality of the processor interface is produced by the co-simulation environment. The method is that the SW simulator emits the event to the bus functional model of the processor, which translates the event into a new state or states of the processor interface.

The main difference between commercial solutions and the MSE is that the MSE does not use the bus functional model for connecting custom hardware and software. The functionality of the processor executing the software is simulated in the software simulation side and the effect of custom hardware in the simulation can be seen as events, not as real processor bus cycles. The advantage of this approach is that the number of signal events caused by the SW simulation in the HW simulation is reduced, which enables faster simulation. The disadvantage of this approach is that the effect of the processor bus and memory references cannot be seen in the HW simulation.

Both EagleI and Seamless offer several ways of optimising the performance of the simulation, e.g. by filtering uninteresting signal events. The filtering speeds up the simulation, but causes, however, some additional workload on the simulation backplane. The MSE approach does not involve signal filtering allowing thus better performance in the inter-simulator communication. These two approaches are depicted in Figure 21. The left hand side of the picture describes the commercial solution, where all other parts except the SW simulator and the simulation backplane are in the HW simulator. The right hand side of Figure 21 describes the OMI/TOOLS approach, where only the hardware of interest is in the HW simulator.

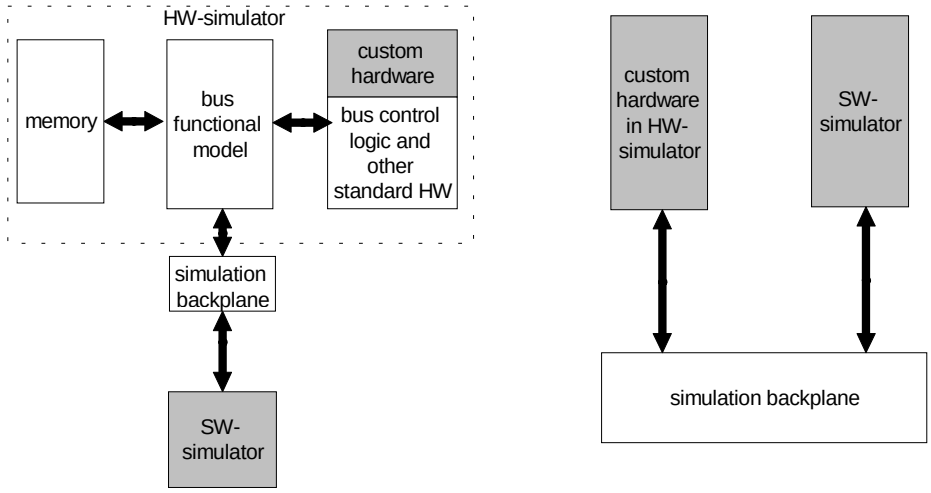


Figure 21. The different approaches of the commercial tools and the MSE.

When comparing the MSE to commercial systems, it can be seen that both approaches have their benefits as well as disadvantages. The commercial tools can be used for validating relatively small amounts of software together with systems hardware. The software may be used as a testbench for hardware design and vice versa during the development. Still, it is unlikely that the accuracy or the comprehension of the testbench is enough for final validation of custom hardware.

The OMI/TOOLS approach allows the designer to run large portions of software together with custom hardware. This allows more comprehensive system level testing, even partially interactive, since the designer has more time to experiment several scenarios. The obtained accuracy is less than in the commercial solutions, if the designer is interested in the processor bus detail functionality and its effects on the system behaviour. This is, however, due to the different target of the simulation. The MSE is best suited for modelling the system with less detail than the commercial co-simulation methods, in order to give much faster simulation. Still the MSE does not prohibit the use of detailed models if required.

6.4 FUTURE WORK

In co-simulation there is always a need for greater simulation speed. This is due to the fact that the gap between real world execution speed and simulation speed is enormous and thus limits the testing only to some case situations. This means that the most interesting and simultaneously demanding issues when developing the MSE are to improve the performance.

As to the performance of the VHDL simulator, there are only very limited possibilities for improvements. The IEI of the VHDL simulator does not cause significant time consumption and the nature of the VHDL simulation gives only little room for increasing the length of the simulation sub-phase. From the modelling point of view, the future work could mainly mean more generic support for datatypes, currently datatype matching is considered to be the designers' job. The more generic handling of the datatypes could enable the composition creation between VHDL-design modules even when the flows are of a different data-type. One interesting possibility could also be the testing of

embedding another VHDL simulator in the MSE, since the methods selected were designed to be easily adaptable to new VHDL simulators.

In co-simulation, there is always a trade-off between the achieved performance and the desired accuracy. Basically, the only way to improve performance without affecting the accuracy is to improve the speed of communication. However, the possibilities to improve the speed of the communication are limited, even with faster network the communication is still the bottleneck. It is highly unlikely that the communication speed could be improved in the near future so that the total simulation speed would increase tenfold. Development of the communication speed goes hand in hand with the development of computer devices and networks, but also the complexity of the designs follow the same trend. This means that over time, the faster communication link and faster workstation is just a necessity if even the current level is to be maintained.

In order to improve simulation performance significantly, the length of the sub-phase should be increased. The longer sub-phase gives more possibilities to run parallel simulations enabling the designer to concentrate more processor capacity in one simulation. The disadvantage is that if more information from the simulation model behaviour is not available, the increasing of the sub-phase length leads to poorer accuracy. There are basically two possibilities for getting more information from the model: from the designer, or by simulating.

The possibility for the designer to configure the minimum sub-phase length could be the easiest way to improve performance. However, it could seriously deteriorate the accuracy if the designer does not know the system thoroughly. One possibility could be to allow certain parts of the simulation to be run with less accuracy and most important parts with higher accuracy. The key factor is, however, that the configuring of the simulation environment should still remain easy and logical.

An interesting possibility would be the use of the data gathered from previous simulation runs or even dynamically from the ongoing simulation. This could allow fast but easy configuring even without sacrificing much accuracy. However, the development of suitable algorithms and the effort required for implementing these in simulation might be excessive.

7. CONCLUSIONS

The design and verification of the future systems-on-a-chip demand ever increasing amounts of simulation. It is beneficial even for today's system development, since behaviour of the system can be verified as a whole, without building an actual prototype. In future, co-verification can be considered as a necessity if shorter time-to-market and first-time-right design are required. Co-simulation as a solution for system verification requires reasonable simulation speed being achieved. Even the current commercial co-simulation methods give valuable benefits by reducing the integration problems. However, real benefits from the co-simulation become available if large amounts of tests and large portions of software can be simulated concurrently with custom hardware.

The objective of this work was to integrate a VHDL simulator to the OMI/TOOLS MSE. The integration means that the VHDL can be used to describe parts of the systems to be simulated with the MSE. This requires that the VHDL can be used as a component in system-level modelling, and during a simulation a VHDL simulator can be controlled via the simulation environment. In order to fulfil these requirements, the Simulation Configuration Generator (SCG) and InteroperationEngine Interface (IEI) were constructed. The former tool takes care the VHDL simulator's interfacing to MSE modelling, and the latter enables the controlling of the VHDL simulator during the co-simulation. Additionally, the SCG enables the composing of the several VHDL-components into one simulatable composition, thus improving the degree of utilisation of one VHDL simulator licence.

Early tests of the OMI/TOOLS MSE revealed that it has the potential to outperform the existing commercial solutions. Simulation performance testing was not the main objective of this work and thus was not tested thoroughly. However, even the artificial test models indicated good performance. This is especially true with large systems, since the MSE solution enables much more communication than existing products between different parts of the system before the communication link is saturated. However, it is important to remember that co-simulation is still very slow when compared to the real-time execution. Improving the performance would allow to interactive testing of embedded systems, for example, calling with a mobile phone, which is still presented as a virtual co-simulated model. Such real-time testing with complex

designs requires much more simulation speed than is achievable today, even with the MSE. This same trend for demands on simulation speed was clearly shown with the questionnaire discussed in chapter 3.3.

The performance figures presented in this work are subject to change since the development of the OMI/TOOLS MSE continues. Nevertheless most of the basic solutions of the simulation environment are fixed, so it is quite likely that the performance figures given are not so very far from the final performance.

REFERENCES

[1] Yu, A. The Future of microprocessors. *IEEE Micro*, 1996. Vol. 16, No. 22, p. 47. ISSN 0272-1732

[2] Heikkinen, M. et al. Requirements analysis and specification for the modelling and simulation environment. OMI/TOOLS-TR-3.1.1-01, VTT electronics, 1996. 84 p.

[3] Soininen, J.-P., Huttunen, T., Saarikettu, J., Melakari, K., Ong, S. A. & Tiensyrjä, K. InCo - An interactive codesign framework for real-time embedded systems. Espoo: Technical Research Centre of Finland, 1998. 206 p. (VTT Publications 344.) ISBN 951-38-5229-6

[4] Mead, C. A. Scaling of MOS technology. *IEEE Micro*, 1996. Vol. 16, No. 22, p. 48. ISSN 0272-1732.

[5] Hartenstein, R.W. The classification of hardware description languages. Amsterdam: Elsevier Science Publishers B.V., 1987 493 p. ISBN 0-444-87897-1

[6] Gajski, D. D., Vahid, F., Narayan, S. & Gong, J. Specification and design of embedded systems. Englewood Cliffs, NJ: Prentice Hall, 1994. 450 p. ISBN 0-13-150731-1

[7] Cyclone Cycle-Based Simulator, Mountain View: Synopsys inc. 1996. 11 p. (technology backgrounder)

[8] Karpel, R. Ultraspec: A Cycle-Based Simulator Engine for VHDL. Marlborough: Viewlogic Systems Inc., 1995. 3 p. (White Paper)

[9] Dreike, P. & McCoy, J. Co-Simulating Software and Hardware in Embedded Systems. *Embedded systems programming* 1997. Vol. 10, No. 6, pp. 12 - 32. ISSN 1040-3272

[10] SimExpress. Wilssonville: Mentor Graphics Corporation 1997. 2 p. (brochure)

[11] The NSIM hardware accelerator. Cupertino: IKOS systems Inc. 1994. 4 p. (brochure)

[12] Arkos Hardware Emulator and Simulation Accelerator. Mountain View: Synopsys Inc, 1996. 9 p. (technology backgrounder)

[13] qaweb.qcktrn.com:80, Quickturn Design Systems Inc. (Quickturn web page)

[14] Honka, H. A simulation-based approach to testing embedded software. Espoo: Technical Research Centre of Finland, 1992. 118 p. (VTT publications 124). ISBN 951-38-4243-6

[15] Jerraya, A. A. et al. Models and Languages for System-Level Specification and Design. In: Staunstrup, J. & Wolf, W. Hardware/Software Co-design: Principles and Practice. Kluwer Academic Publishers, 1997. 32 p. ISBN 0-7923-8013-4

[16] Pulli, P. & Elmstrøm, R. IPTES: A Concurrent engineering approach for real-time software development. *Real-Time Systems*, 1993. Vol. 5, No. 2/3, pp. 139-152. ISSN 0922-6443

[17] www.yokogawa.co.jp, Yokogawa Electric Corporation web page.

[18] Buck, J. Ha, S., Lee, E. & Messerschmitt, D. Ptolemy: A Framework for Simulating and prototyping Heterogeneous Systems. *International Journal in Computer Simulation, Special Issue on Simulation Software Development,* 1994. Vol. 4, pp. 155 - 182. ISSN 1055-8470

[19] Seamless Co-Verification Environment. Wilsonville: Mentor Graphics, 1997. 4 p. (HW/SW co-simulation datasheet)

[20] Eagle Frequently Asked Questions. web page www.viewlogic.com/eagle/faq.htm

[21] Van Rompaey, K., Verkest, D., Bolsens, I. & De Man, H. CoWare - A design environment for heterogeneous hardware/software systems. In: Proceedings of Euro-DAC'96 European Design

Automation Conference with Euro-VHDL'96 and Exhibition, Geneva, Switzerland, September 16 - 20, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996. Pp. 252 - 257. ISBN 0-8186-7573-x

[22] Virtual Socket Interface (VSI) Alliance Membership Agreement. VSI Alliance Inc., Los Gatos, CA, 1997. 4 p. (available at: http://www.vsi.org)

[23] Co-simulation Link for the Hardware Design System. Cadence Design Sytems Inc., 1994. 2 p. (product data sheet)

[24] Heikkinen, M. & Tiensyrjä K. MSE End-User Questionnaire Summary OMI/TOOLS-VTT-032-01. VTT Electronics, 1997. 14 p.

[25] Arundel, P. Preliminary Product Packaging Report, OMI/TOOLS-TR-1.3.2-01. Milano: Etnoteam S.p.A, 1996. 59 p.

[26] Mini SQL A Lightweight Database server, User Manual Release 1.0.11. Golden Coast, Queensland, Australia: Hughes Technologies Pty Ltd., 1996. 19 p. (Available at: http://www.Hughes.com.au/)

[27] Saarikettu, J. Model Interface VTT-041-01b. Oulu: VTT Electronics, 1997. 148 p.

[28] Prosa/SA structured analysis and design environment V4.00. User's manual, Oulu: Insoft Oy. 1994. 129p.

[29] Van Almsick, W. et al. InterOperation Engine Communication Transmission Protocol (ICTP), Version 1.0. Hanover: SIBET GmbH. 1997. 87 p.

[30] Van Almsick, W. et al. IOE/IEI Communication Library (ICL). Version 1.0, Hanover: SIBET GmbH. 1997. 127 p.

[31] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993. New York: IEEE. 1994. 263 p.

[32] QuickHDL User's and Reference Manual V8.5_4. Wilsonville, Oregon, USA: Mentor Graphics Corp. 1995. 197 p.

[33] Granlund, T. GNU MP The GNU Multiple Precision Arithmetic Library V2.02. Stockholm: TMG Datakonsult. 1996. 39 p. (Available at: http://www.gnu.org/)

[34] Leef, S. Klein, R. Hardware, Software join in simulation. *Electronic Engineering Times,* 1996. Issue 904. ISSN 0192-1541

Appendix A: VHDL code generated by SCG

```
library module1
                    -- library declarations for the composed objects

use module1.all



library module2

use module2.all

entity IEI is                                                    -- Entity declaration of the IEI module
        port(
             input1:in integer;
           input2:in integer;
           input3:in integer;
             output1:out integer
             );
end IEI;

architecture behaviour of IEI is          -- Declaration of the foreign attribute
        attribute foreign : string;
        attribute foreign of behaviour : architecture is "FLIpart $MSE_HOME/IEI_of_qhdl/FLIpart.o";
begin
end;
architecture structure of COMBO is                    -- The name of the toplevel

component module1          -- The name of the VHDL module in component declaration
 port(
  signal1:in integer;                  -- Interface to the other modules
  signal2:in integer,                  -- These have to be in correct order
  result:out integer
  );
end component;

component module2                      -- The name of the second VHDL module
 port(
  signal1:out integer;                 -- The names in interface does not have to match
  signal2:out integer;                 -- to the interface of the first module
  feedback:in integer
  );
end component;

component IEI                                                    -- The name of the IEI module
 port(
  input1:in integer;
  input2:in integer;
```

```
  input3:in integer;                    -- There could be any amount of signals (in or out)
  output1:out integer
   );                                                    -- connected in here
end component;

signal signal1:integer;                 -- The inter-module signals
signal signal2:integer;
signal feedback:integer;
signal result:integer;

begin

first:module1 port map(                                 -- The signal mappings to the components
  signal1 => signal1,                   -- The names of the component instances have to be unique
  signal2 => signal2;
  result => result
   );

second:module2 port map(
  signal1 => signal1,
  signal2 => signal2,
  feedback => feedback
   );

third:IEI port map(
  input1 => result;
  input2 => signal1;
  input3 => signal2;
  output1 => feedback
   );

end structure;
```

Appendix B: Measurement results

The co-simulation results measured for the Fibonacci test model are presented in Table B - 1, Table B - 2 and Table B - 3. All the simulation time values are averages measured from three simulation runs. *Degree, D* refers to the degree of the Fibonacci algorithm calculated. This figure determines the load for ClearSim2. *Events, E* refers to amount of inter-simulator events. *Loops, L* defines the number of times the Fibonacci is calculated. *Mtime* refers to model time of the Fibonacci test. *Sim. time* refers to the simulation time of the Fibonacci test. *Instructions, I* contains the total amount of assembly instructions executed. *I/sec* contains the amount of instructions executed on average during one second of simulation. *I/event* is an average of the instructions executed before an event in the VHDL simulator occurs.

Table B - 1. Simulation results of Fibonacci test model. Simulation is executed on one SUN ULTRA 1 workstation with 170 MHz Sparc processor.

D.	E.	L.	Mtime	Sim. time	I.	I/sec	I/E
2	5120	1600	1603508	23	146720	6379	92
4	5120	1600	4381108	24	410240	17093	256
6	5120	1600	11587508	28	1086320	38797	679
8	5120	1600	30435508	37	2854320	77144	1784
10	620	200	9980308	10	935290	93529	4676
12	620	200	26125908	19	2449790	128936	12249
14	620	200	68395508	41	6414790	156458	32073
16	620	200	179058708	96	16795290	174951	83976
18	310	100	234394308	122	21985895	180212	219859
20	310	100	613642708	315	57560395	182731	575604

Table B - 2. Simulation results for Fibonacci test model. Simulation execution is distributed to two workstations. The faster workstation, SUN Ultra 1 with 170 MHz sparc processor, serves as host for the IOE and QuickHDL. The slower workstation, SUN Ultra 1 with 140 MHz processor, serves as host for ClearSim2.

D.	E.	L.	Mtime	Sim. time	I.	I/sec	I/E
2	5120	1600	1603508	21	146720	6986	92
4	5120	1600	4381108	23	410240	17836	256
6	5120	1600	11587508	28	1086320	38797	679
8	5120	1600	30435508	39	2854320	73187	1784
10	620	200	9980308	10	935290	93529	4676
12	620	200	26125908	20	2449790	122489	12249
14	620	200	68395508	46	6414790	139452	32073
16	620	200	179058708	110	16795290	152684	83976
18	310	100	234394308	142	21985895	154830	219859
20	310	100	613642708	360	57560395	159890	575604

Table B - 3. Simulation results for Fibonacci test model. Simulation execution distributed into two workstations. The faster workstation, SUN Ultra 1 with 170 MHz Sparc processor, serves as host for ClearSim2. The slower workstation, SUN Ultra 1 with 140 MHz processor, serves as host for IOE and QuickHDL.

D.	E.	L.	Mtime	Sim. time	I.	I/sec	I/E
2	5120	1600	1603508	23	146720	6379	92
4	5120	1600	4381108	24	410240	17836	256
6	5120	1600	11587508	28	1086320	38797	679
8	5120	1600	30435508	38	2854320	75114	1784
10	620	200	9980308	10	935290	93529	4676
12	620	200	26125908	17	2449790	144105	12249
14	620	200	68395508	41	6414790	156458	32073
16	620	200	179058708	99	16795290	169649	83976
18	310	100	234394308	122	21985895	180212	219859
20	310	100	613642708	315	57560395	182731	575604

The results measured from co-simulation of the heater test model are presented in Table B - 4. All the simulation time values are averages measured from three simulation runs. *Sample* refers to frequency of samples, temperature values, taken from VHDL simulation . *Events* refers to the amount of inter simulator events. *Mtime* refers to model time of the Fibonacci test. *Sim. time* refers to the simulation time of the heater test. *Instruct*. contains a total amount of assembly instructions executed. *I/sec* contains the amount of instructions executed on average during one second of simulation. *I/event* is an average of the instructions executed before an event in the VHDL simulator occurs.

Table B - 4. Simulation results for the heater test single workstation used. The workstation is SUN Ultra 1 with 170 MHz processor.

Sample	Events	Mtime	Sim. time	Instruct	I/sec	I/event
2000 ns	5040	10 ms	60	245775	4096	48
7000 ns	1340	10 ms	28	63361	2262	47
12000 ns	916	10 ms	23	42784	1860	47
20000 ns	568	10 ms	20	26135	1306	46