UNIVERSITY OF SOUTHAMPTON

SystemC-A: Analogue and Mixed-Signal Language For High Level System Design

by

Hessa Jassim Al-Junaid

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

May 2006

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

SystemC-A: Analogue and Mixed-Signal Language For High Level System Design

by Hessa Jassim Al-Junaid

In the light of the growing popularity of mixed, analogue and digital ASICs and System on Chip, several high level hardware description languages (HDLs), such as VHDL and Verilog, have recently been extended to provide analogue and mixed-signal (AMS) modelling capabilities. SystemC is a new language added recently to the existing HDLs used by the digital electronic design community. This research has developed a new methodology that enables the extension of SystemC to the analogue domain and allows simulations of mixed-signal and mixed-domain systems on arbitrary levels of abstraction. The developed AMS extension is named SystemC-A and complies with SystemC semantics. In many respects, SystemC-A is more powerful than many existing HDLs.

The contributions of this research can be summarised as follows: Firstly, new syntax elements and classes that extend SystemC to the analogue domain have been developed. The new language construct elements support analogue system variables, analogue components and user defined equations. In addition to the various abstraction levels provided by SystemC, the developed extension provides extra abstraction levels which are specific to analogue systems. A numerically efficient analogue kernel has been developed and implemented in which a novel equation formulation method for nonlinear algebraic and differential equations (DAEs) is developed.

Secondly, a novel mixed-signal synchronisation method to integrate the analogue kernel with the digital one has been developed. The implementation of the lock-step synchronisation method provides an efficient handling of extremely small and zero time step sizes and enables analysis with arbitrary accuracy. Support for digital-analogue interfaces has been provided for easy and smooth integration of digital and analogue parts.

Finally, SystemC-A is validated and optimised using a suite of numerically difficult analogue, mixed-signal, and mixed-domain examples. Their complexity ranges from simple sets of DAEs to highly complex mixed-signal systems, which are difficult to handle by existing HDLs. SystemC-A supports different types of continuous-time analysis suitable for mixed-signal modelling. For example, it supports large-signal time domain noise analysis, which is traditionally difficult to implement in a mixed-signal context.

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

ABSTOL	**Abs**olute **To**lerance
A/D	**A**nalogue/**D**igital
ADC	**A**nalogue to **D**igital **C**onverter
AHDL	**A**nalogue **H**ardware **D**escription **L**anguage
AMS	**A**nalogue and **M**ixed-**S**ignal
AMS HDL	**A**nalogue and **M**ixed **S**ignal **HDL**
ANSI	**A**merican **N**ational **S**tandards **I**nstitute
ASIC	**A**pplications **S**pecific **I**ntegrated **C**ircuit
BDF	**B**ackward **D**ifferentiation **F**ormula
CAD	**C**omputer **A**ided **D**esign
D/A	**D**igital/**A**nalogue
DAC	**D**igital to **A**nalogue **C**onverter
DAE	**D**ifferential and **A**lgebraic **E**quation
DSP	**D**igital **S**ignal **P**rocessing
EB	**E**rror **B**ound
EDA	**E**lectronic **D**esign **A**utomation
GUI	**G**raphical **U**ser **I**nterface
HDL	**H**ardware **D**escription **L**anguage
IC	**I**ntegrated **C**ircuit
IP	**I**ntellectual **P**roperty
ISS	**I**nstruction **S**et **S**imulator
KCL	**K**irchhoff's **C**urrent **L**aw
LRM	**L**angauge **R**eference **M**anual
LTE	**L**ocal **T**runcation **E**rror
MEMS	**M**icro **E**lectro **M**echanical **S**ystem
MNA	**M**odified **N**odal **A**nalysis
MoC	**M**odel **o**f **C**omputation
NR	**N**ewton **R**aphson
ODE	**O**rdinary **D**ifferential **E**quation

OO-NQN	**O**bject **O**riented **N**ewton-**Q**uasi **N**ewton
OSCI	**O**pen **S**ystem**C** **I**nitiative
PDAE	**P**artial **DAE**
PLI	**P**rogramming **L**anguage **I**nterface
PLL	**P**hase **L**ocked **L**oop
PWM	**P**ulse **W**idth **M**odulation
RELTOL	**Rel**ative **Tol**erance
RF	**R**adio **F**requency
RHS	**R**ight **H**and **S**ide
RTL	**R**egister **T**ransfer **L**evel
SCV	**S**ystem**C** **V**erification library
SMPS	**S**witched **M**ode **P**ower **S**upply
SoC	**S**ystem **o**n **C**hip
TBV	**T**ransaction **B**ased **V**erification
TLM	**T**ransaction **L**evel **M**odelling
VCO	**V**oltage **C**ontrolled **O**scillator
VHDL	**V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage
VHDL-AMS	**VHDL** for **A**nalogue and **M**ixed **S**ignal
WGN	**W**hite **G**aussian **N**oise

Acknowledgements

I would like to express my thanks and appreciation to my supervisor DR. Tom Kazmierski who has provided the brilliant ideas which are the mainframe of this research, and precious guidance both in academy and life, when I needed it most.

Also, I would like to acknowledge and express my appreciation to Prof. Bashir Al-Hashimi for being the examiner for both the nine-month report and the MPhil transfer. I am grateful for his comments and suggestions on my work.

I wish to acknowledge the School of Electronics and Computer Science ECS in the University of Southampton for the high standards of materials and computing facilities and for funding me to attend overseas conferences. A special gratitude to all the staff of the ECS who supported us as postgraduate students after the massive fire in Mountbatten building. Thanks to all my colleagues and friends at the Electronic System Design group for their continuous willingness to help.

I would like to express my thanks to the University of Bahrain for the scholarship and the funding during my PhD.

I am forever thankful to my husband Ebrahim Al-Gallaf for his love and strong encouragement to pursue my studies.

To my baby Nayef

Chapter 1

Introduction

Design complexity and demanding time-to-market constraints have lead to considerable challenges in the development of electronic design methodologies and Computer Aided Design (CAD) tools. Furthermore, the integration of a complete complex System on a single Chip (SoC) has begun a new era [1, 2, 3]. SoC has created a need to powerful CAD tools and methodologies which would be integrating information from multiple heterogenous sources (analogue parts, processors, RAM, ROM, etc.) and have the ability to work at high levels of abstraction [4].

On the other hand, advances in integrated circuit technology have been the driving force behind the extensive development of digital Hardware Description Languages (HDLs), whilst Analogue and Mixed-Signal (AMS) high level modelling is lagging behind the design community with immature design methodologies [5, 6]. This has created a gap in the design of the two different parts which threaten the rate of production. Despite the success of digital systems, analogue circuitry is still needed in particular in modern ASIC (Applications Specific Integrated Circuit) designed for telecommunication, wireless and computer network systems [1, 5]. The design of analogue blocks in SoC (e.g. Figure 1.1) and ASIC is still done to a large extent manually which requires time and effort together with specific skills

[7]. All of these advances and challenges have put a pressure on CAD of AMS systems to keep up with the success of pure digital CAD.

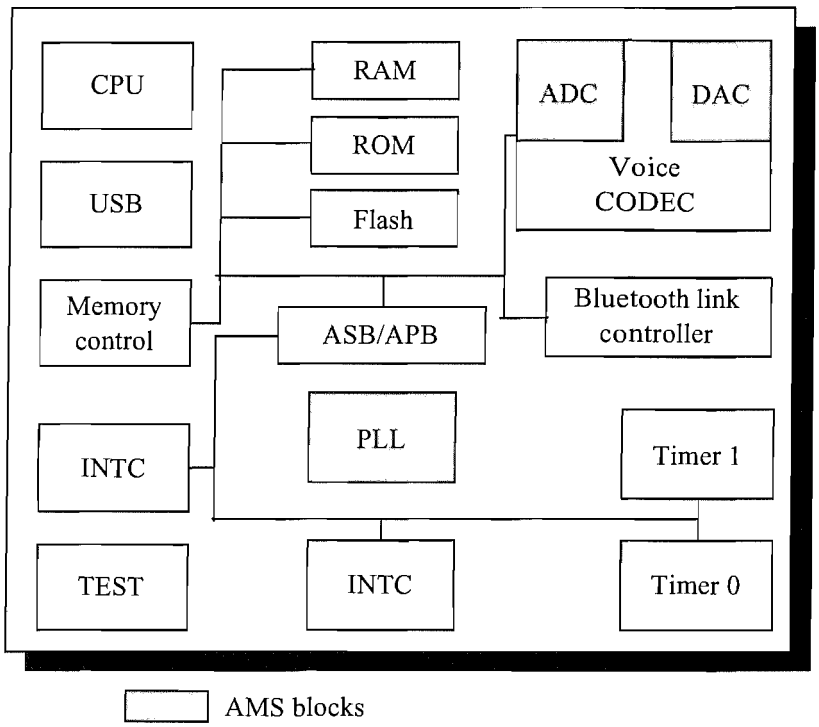

FIGURE 1.1: Example of a mixed-signal System on Chip in telecommunications [1].

The Electronic Design Automation (EDA) industry and academia were trying extensively to meet these needs following different approaches [2]. One common approach is to model and simulate digital and analogue systems with digital HDLs and analogue design tools respectively. Digital HDLs such as VHDL [8], Verilog [9], SystemC [10] and SystemVerilog [11] are used to model digital systems while analogue design or general purpose equation solving tools such as SPICE [12] and MATLAB [13] dominate in the modelling of analogue systems at different abstraction levels. Another approach is to extend classical HDLs intended originally to model digital systems to model analogue systems such that both parts are modelled and simulated in a single environment such as VHDL-AMS [14] and Verilog-AMS [15]. A third approach is to readapt software programming languages

such as C/C++ [16] to model analogue and digital hardware. It can be accomplished by adding special language constructs for hardware description and timing and defining hardware description semantics.

The recent trend in digital modelling is toward C/C++ based modelling [17, 18] either through libraries or abstractions. C/C++ is already in use by hardware engineers at algorithmic level to estimate the system performances and verify the functional correctness of the design. There are various C/C++ based HDLs provided by EDA suppliers such as SystemC [10] and SystemVerilog [11], and from universities such as Handel-C [19] and SpecC [20] which then have been moved to the EDA tool providers.

Of all the C/C++ based HDLs, SystemC [10] was the focused HDL since its very beginning. The first version of SystemC was released on September 1999 and since then it has gained a wide acceptance and support from industry. This broad acceptance suggests that the Open SystemC Initiative (OSCI) met an important need with the right approach. SystemC is a standardised modelling language intended to enable system level design and Intellectual Property (IP) exchange at multiple abstraction levels for systems containing both software and hardware components and can work as an alternative to existing HDLs [21].

The latest SystemC version (V2.0.1) has been in use since 2003. Although it resembles the existing HDLs and adds more features for high level digital modelling, it does not support AMS modelling yet. This lack of analogue modelling capability and the popularity and reliability of SystemC provided the main motivation for this research (see Section 1.5).

Within this context, this chapter is organised as follows: in Section 1.1 a broad view on system design stages (which is called top-down IC design methodology) is given illustrating the disadvantages of the use of different languages and tools at different levels of abstraction. Section 1.2 explains in detail the potential of SystemC which

motivate its use in this research. Section 1.4 defines AMS modelling and identifies its potential applications and the most important challenges to be tackled. Section 1.5 presents the main objectives and contributions of this work and Section 1.6 describes the challenges behind the choice of the case studies in order to validate SystemC-A. Finally, Section 1.7 outlines the structure of the thesis.

1.1 System Design Methodology

Top-down methodologies in Integrated Circuit (IC) design have been used for complex design tasks in many disciplines for a number of years [6]. Digital top-down design is supported by many simulation levels (e.g. behavioural, RTL, logic, etc.) which has facilitated model generation at various steps of the design process. With the availability of digital HDLs, this model generation process has been greatly simplified and reduced the number of simulators needed to support top-down design methodology for digital circuits.

In the past, AMS designers had great difficulty in top-down design due to the lack of modelling tools and languages [22]. Typically, a transistor level netlist description would have to be generated for each block in a design at each intermediate step in the top-down process. This usually resulted in little or no simulation support early in the design cycle with the result that conceptual or specification errors were not detected until later in the design cycle. This could have a serious impact on the design schedule as errors not detected early in the design cycle may force significant redesign of all or part of the circuit.

The recent evolution of AMS HDLs made top-down design of AMS circuits feasible. The customised top-down approach of AMS systems to the design flow in Figure 1.2 uses the following major levels of abstraction [23], where conceptual/algorithmic level modelling serves as the highest abstraction level.

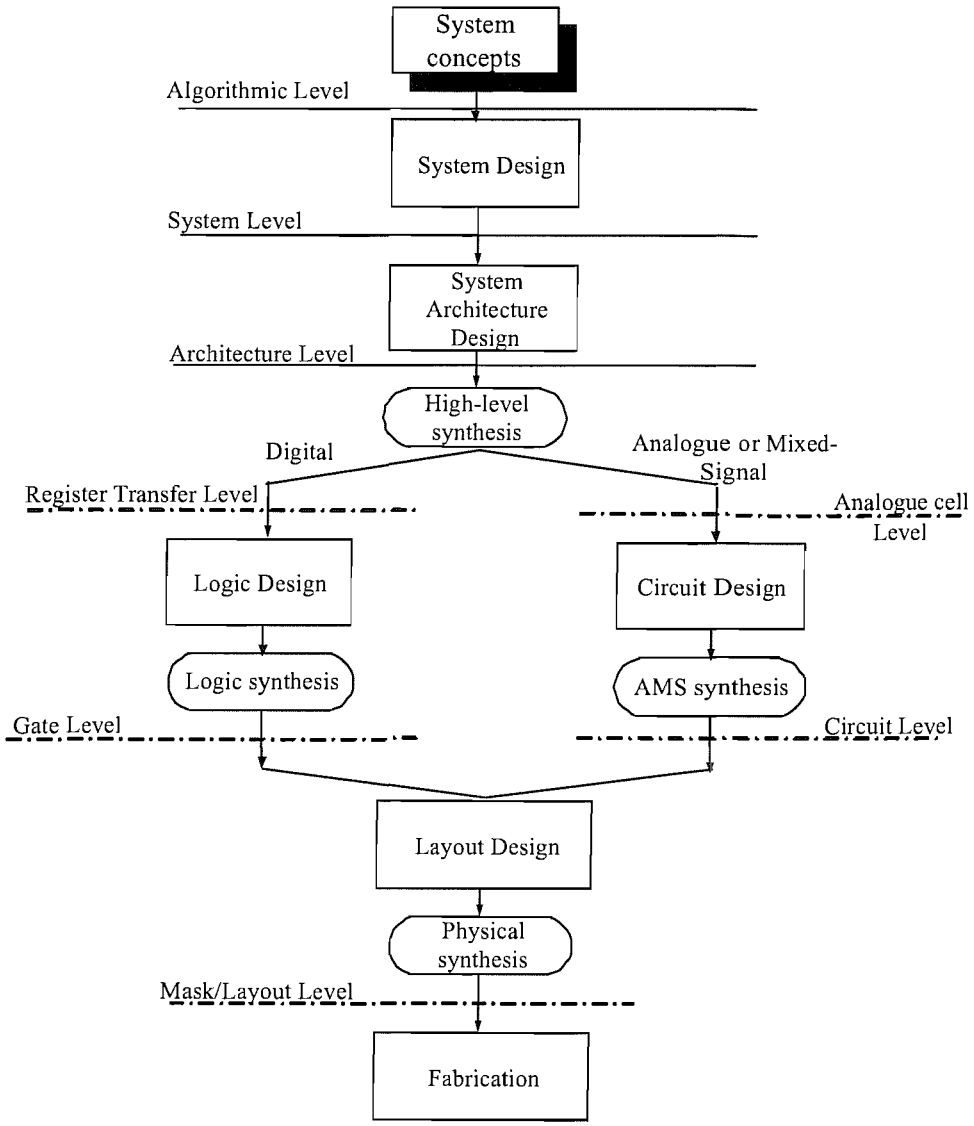

FIGURE 1.2: Design flow and levels of modelling abstractions.

- *Conceptual or algorithmic level*, is where the specification of the design is mainly represented by signal flow diagrams with blocks described by mathematical equations. No structural details are considered. Designers can simulate at this level to prove the basic concepts of the system, and build a set of specifications for structural implementation. The system is mainly modelled at this level using C/C++ programming languages and sometimes application specific descriptions (such as Matlab/Simulink, Mathematica).

- *System level*, this is the first stage of the actual design, where the overall architecture of the system is designed and partitioned. Hardware and software parts are defined and both are specified in appropriate languages. In addition, the interfaces have to be specified. The hardware components are described at the behavioural level (for the analogue part, blocks are described by Differential and Algebraic Equations (DAE) and/or s-domain transfer functions and for the digital part blocks are described by difference algebraic equations and/or z-domain transfer functions). The system level partitioning and specifications are then verified using detailed co-simulation techniques.

- *Architectural level*, is a high level decomposition of the hardware part into an architecture consisting of functional blocks required to realise the specified behavioural description. Also, this level includes partitioning between analogue and digital blocks. The specifications of the various blocks are defined and described in an HDL (e.g. VHDL and VHDL-AMS). The high level architecture is then verified against the specifications using behavioural simulations.

- *Register Transfer Level* (RTL), is the highest structural level, where generic functions and variables are replaced by structural blocks. Blocks are defined as collections of circuits that store data and circuits that operate on data (storage and operators) (e.g. a variable is replaced by a register of a given size; an IF-statement is represented by a multiplexer, arithmetic operations are replaced with Arithmetic Logic Units (ALUs)). Simulation at this level verifies the logic definitions for the operators and sometimes verifies critical parameters like the clock frequency at which data is passed from one storage block to another.

- *Gate level*, is a structural level where components are described in terms of digital primitives (Boolean logic gates with timing data). Storage and

operators are broken down into the digital functions that implement their function (e.g., a 2-to-1 multiplexer is represented by two AND gates and an OR gate). Timing of individual signal paths can be verified at gate level simulation.

- *Analogue cell and circuit levels*, are the structural levels for the analogue blocks. In the selected technology process, the model at these levels represent a fully sized device level circuit schematic (with basic elements such as transistors, diodes, resistors, and capacitors). The resulting circuit of design is then verified against the specifications using SPICE-like circuit simulators. In cell level an equivalent circuit representation (macromodel) might be used to approximate similar behaviour of the original circuit in order to speed up analogue simulators.

- *Layout level*, represents the layout of an IC which includes block placement, routing, and power-grid routing. Crosstalk and substrate coupling analysis are important in mixed-signal ICs. Proper test structures are inserted to make the IC testable. Detailed verification (e.g. timing analysis) is performed. At this level, the system is verified by co-simulating the hardware part with the embedded software. Finally, masks are generated and the ICs are fabricated. Different testing techniques are performed during and after fabrication in order to reject faulty ICs.

From the initial specifications to the final chip, the design goes through a number of translation and verification steps. The translation of the descriptions from one level of abstraction into the other is referred to as *synthesis*. *Verification* is used to check whether the design in the current level of abstraction is correct, conforming with the specifications. Both steps are performed using current HDLs.

There are a number of problems with the top-down design approach which arise from using different specification languages [21]:

- Manual conversion from C/C++ to HDL: the designer creates and verifies the specification of his model using C/C++, and then translates the design manually into an HDL. This conversion is necessary because the logic synthesis tools used by most equipment and IC manufacture require RTL HDL input. This process is very tedious and error prone and leads to a double verification load.

- Separation between system model and HDL model: once the model of the system is converted to HDL, the HDL model becomes the focus of development and the C/C++ model becomes out of date.

- Different testbenches: testbenches that are created to validate the C/C++ functionality typically cannot be used to validate the HDL model. Thus, testbenches needs to be converted from C/C++ to HDL.

Using top-down design methodology is expected to become the norm for designing mixed-signal circuits [24]. Using SystemC as a platform, on which the AMS extension will be developed, can enhance the top-down design methodology evolution for AMS systems. Being a system level language as well as an HDL, the SystemC design approach covers most of the design cycle and offers many advantages over the traditional design cycle, including the following [21]:

- Refinement methodology: the design does not have to be converted from a C-level description to an HDL in one large effort. The design is refined in small sections to add the necessary hardware and timing constructs to produce a good design. Also, the designer can easily implement design changes and detect bugs during refinement.

- Testbenches reuse: testbenches created at higher levels can be reused in lower levels of the design saving conversion time. Testbenches created to validate the system level can be used to test RTL design.

- Unified language: using SystemC enables the designer to utilise one language for most of the design cycle, a language for hardware/software and analogue/digital designs at different abstraction levels.

Many EDA products built on SystemC have been announced [25, 26]. These products support modelling and specification of digital systems which is the first step in the design cycle. Some products took a step further and serve as an aid in hardware synthesis from SystemC [27].

1.2 Why SystemC?

An EDA survey [28] has shown that more than 80% of the responding designers are using or planning to use C/C++. Another interesting result worth mentioning is of a worldwide online EDA survey conducted on November 2003 on design trends [29]. The survey respondent's backgrounds were, hardware engineers, system engineers, verification engineers and others. The survey has shown that the use of SystemC is expected to grow 3 times by 2004. Only 4% of the respondents currently use SystemVerilog. Handel-C also shows a significant growth, as an implementation language from C. ANSI C modelling usage is expected to stay relatively flat as models are done in a system language whereas VHDL is expected to lose a small amount of ground to system languages.

This trend towards C/C++ based HDLs and the big support of SystemC in the industry have provided the motivation to use SystemC in this research as a system language and an HDL platform in which an AMS extension and new requirements will be added. C/C++ based HDLs offer many advantages as opposed to other HDLs [30, 31], some are summarised below:

Hardware/software migration

Because of the growing amount of software running on any system, many of today's chips incorporate processor cores running instruction codes compiled from programming languages, in particular C [32]. Thus, it is possible to move certain functions that are implemented in hardware to software instead of forcing an implementation as an entirely dedicated hardware-based circuit. By doing so, faster simulations become available [31].

Rich legacy code

Moreover, among programming languages, C/C++ have been the most widely used in the last two decades. As a result there is a vast amount of legacy code and libraries that can be reused to quickly model systems. Based on its nature, C/C++ supports reusability of design descriptions [30].

Hardware/software unified language

It has been observed [30] that system and software engineers tend to use C/C++ while their hardware counterparts are using HDLs such as VHDL and Verilog, and from the use of different description languages problems are arising. These problems can be solved by the adoption of a common C++ style for hardware and software parts of the system thus eliminating the Programming Language Interface (PLI) overhead.

Furthermore, using SystemC enables designers to stay at the C-level for most of the implementation cycle and serves as a single environment for electronic architects, verification and implementation engineers. In other words, SystemC facilitates the integration and verification of hardware and software (co-verification) components within one environment. The impact of co-verification is that it provides higher

simulation speed. In one case, IC simulation time was slashed from 20 days to only 8 hours [31].

Development efficiency

There are numerous advantages of using C/C++ based tools, as limited resources in terms of design personnel and capital can be invested in the design cycle. If C-based tools are used, there will be no need for specific compiler or specific training when hiring designers. Many engineers already have experience with C/C++ and for those that do not, training takes a shorter time and is available from many sources.

C/C++ is also invaluable in reducing the man-hours needed for coding. The amount of coding is about 10% that of HDL [31], which means that projects previously requiring 10 designers can now be handled by one only. This is possible because C/C++ supports a higher level of abstraction in coding.

Higher abstraction levels

Many people in the industry have complained about the slowness of existing circuit simulation at RTL [2]. The reason is simply that RTL is too low as an abstraction level to start designing multimillion gate systems. In most HDLs, descriptions are done at the RTL, which includes the concept of timing. In C/C++ based HDLs, however, the higher level of abstraction means that the specification can be made at the behavioral level or system level, describing only the functions ignoring circuit details. Furthermore, SystemC introduces one more abstraction level, called the Transaction Level (TL), higher than the RTL and lower than the system level [33]. SystemC is more likely to comply with new advances in the electronics industry and boost the simulation speed [29] as shown in Figure 1.3.

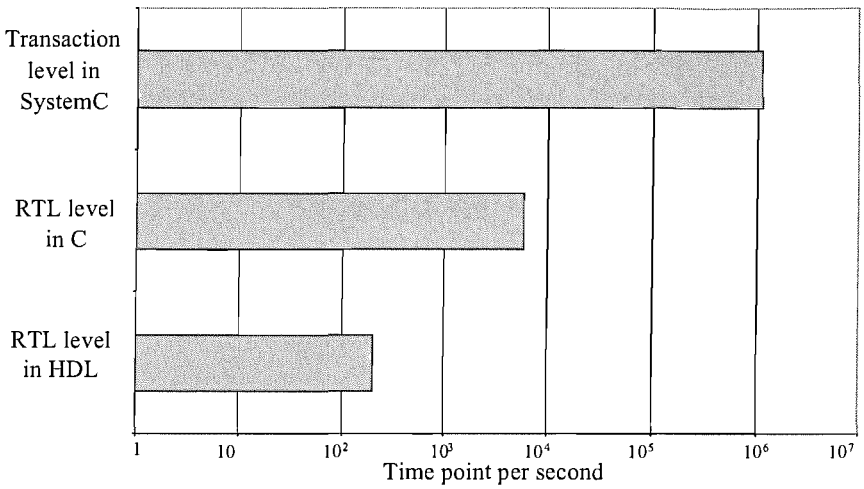

FIGURE 1.3: Increasing abstraction level boosts simulation speed [29].

Electronics industry support

The industry is behind the new trend towards C/C++ based tools. The start and development of SystemC were established through the cooperation of a group of leading electronics companies [34]. The industry support comes from EDA vendors, IP providers, semiconductor, system design and embedded software companies. The electronics community is becoming rich enough with SystemC-based different design and simulation environments [26, 25], verification tools [35, 26] and synthesis tools which cover most of the design cycle [27].

Improved simulation time

For hardware applications, C/C++ have been used often to accelerate the design process since it can be efficiently compiled onto today's architectures and thus used to develop fast simulation models, since C code executes much faster than Verilog/VHDL code [32]. Other factors mentioned above such as better tools for verification, higher abstraction levels and shifting software functions to hardware will lead towards better simulation time.

Non-traditional standardisation

The Open SystemC Initiative (OSCI) [10] is a step away from the traditional approach to establishing standards. One of the beauties of the open source process is that a great deal of emphasis is placed on those who contribute the most high quality content to the technical forum, making even political trade-offs merit-oriented [36]. Another key benefit is speed of validation and adoption. With SystemC, the entire licensee base can immediately download, assess and modify a live, executable version of the standard, whereas with most traditional standards processes it can take years to produce the specification, followed by another year or two of adoption.

More advantages are contained in Chapter 2 when surveying SystemC and also in Appendix B which gives a literature review of SystemC applications.

1.3 SystemC-A versus VHDL-AMS

SystemC-A is aimed to be a complementary language to existing HDLs. However, in many respects SystemC-A is more powerful than VHDL-AMS, the most popular HDL. This section summarises the main advantages of SystemC-A over VHDL-AMS which will be realised in detail throughout the thesis.

SystemC-A could reuse the heritage libraries of C/C++ to shorten developing new libraries such as those for optimisation and signal processing, while in most VHDL-AMS simulators, math libraries are simple and need further additions. Also, while the type of standardisation process of VHDL-AMS had a prolong effect on its development, SystemC's standardisation process has strengthen its development.

With regards to its new methods, SystemC-A includes an equation formulation method which gives the user the choice of simplicity in coding or faster simulation

speed. This equation formulation is unique to SystemC-A user and is not available in VHDL-AMS simulators. Also SystemC-A model has the advantage of faster simulation times which relate to some factors such as the synchronisation method and equation formulation method. Further SystemC-A demonstrate the ease of developing any kind of analysis. For instance, it supports time domain noise analysis which is not supported in some recent analogue simulators. On the other hand, SystemC-A benefits from the advantages of SystemC digital simulator such as the ability to co-model and co-simulate hardware and software in the same environment. Also the ability to model and simulate at high levels of abstraction such as the TLM while VHDL does not support modelling higher than RTL.

1.4 Analogue and Mixed-Signal Modelling

In the past, several approaches have been used to model AMS systems. One approach is to use a circuit simulator and model the digital components at a functional level [37]. Another approach is to use a logic simulator and model the analogue components at a functional level [38]. A more accurate approach is to use a circuit simulator for the analogue components linked to a logic simulator for the digital components [9] which requires tool couplings.

The current approach in mixed-signal modelling is using AMS HDLs such as VHDL-AMS and Verilog-AMS. AMS HDLs describe the behaviour and structure of AMS systems using special language constructs, solving many modelling and simulation difficulties. However, AMS HDLs still need extra modelling effort to model at system level. Their performance in verification of a complex SoC is too low. This is due to the limitation of modelling software and hardware together. HDLs lack communication abstractions found in SystemC that make it possible to model at higher level of abstraction. On the other hand, any further development

in HDLs needs standardisation and this is considered a time consuming process. For example, the development and standardisation of VHDL-AMS took more than 10 years.

Issues related to synchronisation and interfacing between the analogue kernel, with its tiny integration steps, and digital kernel, with its events, are important and determine the efficiency of the overall simulation [23]. The speed of mixed-signal simulation is limited by the inherent speed disadvantage of analogue simulation. It varies according to factors such as simulation setup times, clock frequency, nonlinearities, part size, activity in the digital circuit, and time constraints in the analogue circuit. Mixed-simulator solutions take advantage of stable, highly-tuned algorithms and the user's experience in optimising and iterating simulation runs. The event-driven algorithms in a native mixed-signal simulator are typically not as efficient as those in the state of the art digital simulators, so they sacrifice speed and efficiency.

Rising to meet the growing needs of their users, EDA vendors have made great advances in creating tools that are easy to use while still providing the accuracy essential for design analysis. Today mixed-signal simulation tools are as available as those used exclusively for analogue or digital designs and has led to dramatic improvement both in design quality and in time to market. The developed SystemC-A language is aimed to take a large share of the suggested map shown in Figure 1.4 along with the current analogue simulators and HDLs.

1.4.1 Potential Applications and Challenges

AMS modelling is used for a range of applications in the electronic industry such as amplifiers, data converters, RF circuits, filters and reference generators. Also in other areas such as the automotive [39], biomedical [40], mechanical [41], and other mixed-physical domain areas.

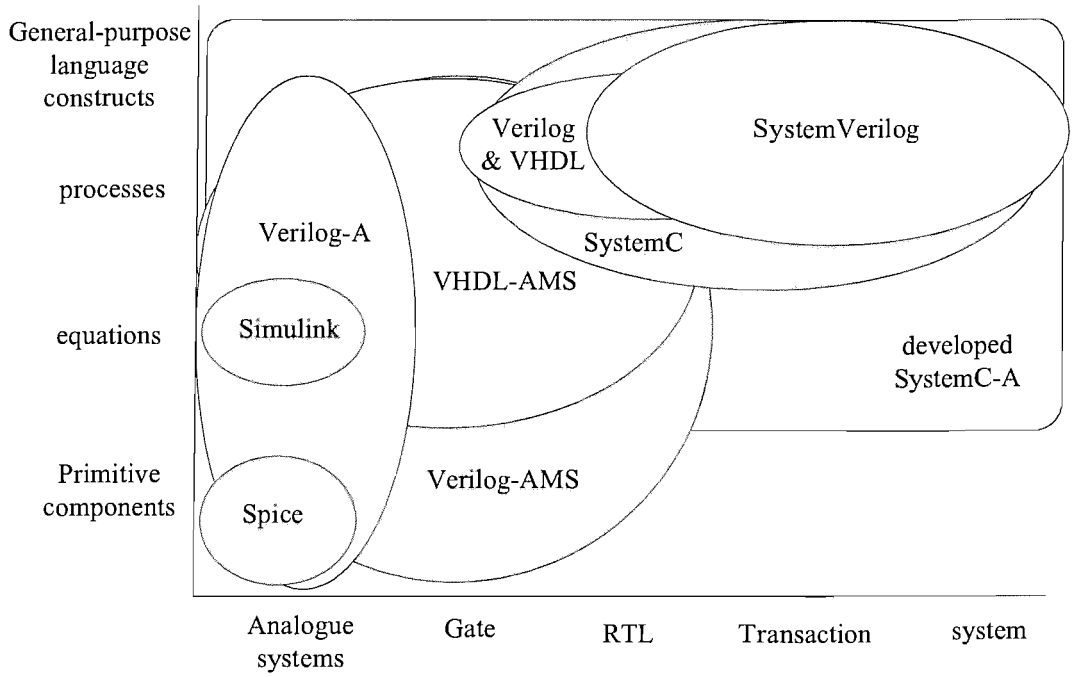

FIGURE 1.4: A suggested map of SystemC-A with the current analogue simulators and HDLs classified based on abstraction levels and design units.

According to the 2003 edition of the International Technology Roadmap for Semiconductors (ITRS) [42], today's SoCs are increasingly mixed-signal designs and facing a number of problems and challenges in the design methodologies and flows, design productivity, modelling, simulation, and verification. The modelling and simulation problems are of great concern in this research, which is trying to find solutions to enhance the process of modelling AMS systems.

The main challenge is the need for higher levels of abstraction to describe AMS systems. There are three reasons for this [23]. At higher levels of the design methodology, the need for higher level models is necessary to describe the pin-to-pin behaviour of the circuits rather than the internal structural implementation. Second, when using analogue IP macro-cells in a SoC context, the virtual component has to be accompanied by its executable model that efficiently models the pin-to-pin behaviour. This model can then be used in SoC design and verification, without knowing the detailed circuit implementation of the virtual component. Third, the verification of mixed-signal systems is computationally too complex to

allow a full simulation of the entire design in practical terms. Therefore, higher description levels of the analogue sections would be extremely helpful.

The above three problems can be solved by adopting modelling paradigms and languages from the digital world in the analogue domain. For example, behavioural and functional simulation levels have been developed for analogue circuits besides the well known circuit level. The main requirement of a future SoC HDL is to be a true mixed-signal, multilevel, mixed-domain simulator with special emphasis on system level design.

The main difficulty with higher level analogue modelling (and not to be solved in this research) is the automatic characterisation of analogue circuits, in particular the automatic generation of analogue macro-models or behavioural models from a given design [23]. This problem needs to be addressed in the near future, as it might be the biggest hurdle to the adoption of high level modelling methodologies and AMS HDLs in industrial designs.

SystemC-A could be an input specification for the synthesis process of analogue circuits. The acceptance of digital synthesis from SystemC could motivate a new synthesis methods for SystemC-A. In the procedure of analogue synthesis, SystemC-A could benefit from the huge existing libraries of optimisation in C/C++ such as genetics algorithm and simulated annealing. Although synthesis of analogue circuits is not well established for VHDL-AMS, methods used in VHDL-AMS can migrate to SystemC-A.

1.5 Research Objectives and Contributions

The main objective of this research is to extend the capabilities of SystemC to model AMS systems at a high level of abstraction and to validate the developed

methods with suitable examples of mixed-signal mixed-domain systems. The developed system level language and the AMS extension to SystemC should be fully derived from SystemC and comply with its semantics. The main contributions out of this research are:

1. **Analogue and Mixed Signal AMS extension of SystemC**

   The AMS extension includes the following:

   New syntax elements and classes to extend SystemC to the analogue domain: Modelling of an analogue system requires a set of differential and algebraic equations (DAEs). DAEs should be easy to define, automatically built and updated, and then numerically solved. For this purpose new language constructs have been developed, such as system variables, circuit components and user defined equations.

   High level nonlinear equation formulation method: A numerically efficient analogue kernel has been constructed in order to simulate analogue systems and also to be linked to SystemC digital kernel. Within the analogue kernel, a novel high level equation formulation method has been developed based on the object-oriented feature of C++.

   Support for various abstraction levels: In addition to modelling at various abstraction levels provided by SystemC digital platform, the proposed extension provides extra abstraction levels which are specific to analogue systems. Although the main focus was on the system level to tackle complexity, the extension is capable of modelling at the netlist level and at analogue behavioural level.

2. **Mixed-signal synchronisation method:** A synchronisation algorithm is necessary when integrating an analogue kernel with a digital one. It is needed to synchronise the numerical integration time-stepping engine with the event-driven paradigm of the digital kernel. A new implementation of

the lock-step method, with efficient handling of zero step-sizes has been used for this purpose. Digital-analogue interfaces are also defined for easy and smooth modelling and simulation of the analogue and digital parts.

3. **Modelling and simulations of complex case studies:** A wide range of analogue and mixed-signal examples has been used to validate the developed SystemC-A. Examples range from small sets of DAEs such as Lorenz chaos model and Van Der Pol oscillator equation to non-trivial mixed-signal systems such as switched mode power supply and phase-locked loop. Complex non-electrical case studies such as ferromagnetic hysteresis and mixed-domain automotive vibration isolation systems also have been modelled. Section 1.6 lists the descriptions and challenges of the chosen case studies.

   Numerically efficient implementation of continuous-time analysis for system level modelling: Necessary continuous-time analysis suitable for mixed-signal system level modelling have been implemented. Examples are the computation of the quiescent state of the system (operating point) as well as mixed-signal transient and noise analysis. Noise analysis is of potential importance as, in a mixed-signal context, it is difficult to implement with traditional circuit simulators as it must be evaluated in the presence of large signal behavior.

The research contributions of Chapters 3-8 are published or under revision, they are listed in Appendix A.

1.6 Descriptions and Challenges of the Chosen Case Studies

The case studies have been carefully chosen to provide different modelling and simulation challenges for the validation of the new methods of SystemC-A. Most of the case studies are benchmarks which have been used in Southampton Validation Suite [43] to validate VHDL-AMS. Table 1.1 shows the different case studies with their challenges.

TABLE 1.1: Descriptions and challenges of the chosen case studies.

Case study	Descriptions and challenges
Van Der Pol Oscillator	- popular, easy to compare, analogue system represented by nonlinear ODE. - modelled at behavioural level. - uses initial conditions. - one single controlled parameter may change the nature of oscillations from sinusoidal to relaxation.
Lorenz Chaos	- popular, easy to compare, analogue system represented by nonlinear ODE. - modelled at behavioural level. - uses initial conditions. - simulated output never reaches a steady state.
Switched Mode Power Supply SMPS	- complex, mixed-signal, nonlinear system. - disparate time scale of its transients. - has some difficulties when modelled in existing analogue simulators. - needs excessive CPU times when modelled at circuit level. - modelled at multiple abstraction level in the same model. - make use of the following SystemC-A constructs: analogue circuit components, D/A interfaces, different module connections.
Phase Locked Loop	- almost all what mentioned for SMPS. - high level VCO model uses the output phase as system variable. - design concerns of noise or jitter performance.
Ferromagnetic Hysteresis	- non-electrical system. - widely used model especially in SPICE and SABER. - model suffers from convergence problems, long analysis time and numerical instability.
Automotive Vibration Isolation	- state of the art in automotive suspension industry. - mixed-domain system of complicated mixed electrical-mechanical hydraulic domains. - involves complex nonlinear DAEs. - involves complex control systems.

1.7 Thesis Structure

The thesis is divided into three phases of nine chapters. The first phase is Chapter 1 and 2 which are mostly background and literature review. The second phase is Chapter 3, 4 and 5 which represent the novel and original core of SystemC-A. The third and final phase Chapter 6, 7 and 8 is modelling and simulation of a wide range of examples to validate the developed syntax and methods of SystemC-A. The research conclusion and future work are given in Chapter 9. The following is a chapter by chapter explanation with each chapter's main points and contributions.

Chapter 2 overviews SystemC language and explains the approach of modelling hardware in a software language. Further, it gives an overview of all modelling attempts of AMS systems in SystemC or C/C++ from the literature. The chapter also reviews VHDL-AMS which has been an inspiration to this research.

In Chapter 3, new methods are developed to facilitate modelling of analogue systems in an AMS environment in an accurate and efficient way. The methods include constructs to represent analogue system variables, analogue components to be modelled at lower or higher abstraction levels and analogue-digital interfaces.

The purpose of Chapter 4 is twofold. The first aim is to overview the state of the art in numerical techniques for AMS simulation. The second aim is to build an efficient analogue kernel which is integrated with the SystemC digital kernel for solving analogue systems described by a set of DAEs. A novel object-oriented nonlinear equation formulation method is developed.

Chapter 5 presents a novel synchronisation method to synchronise the developed analogue kernel with the SystemC digital kernel. The synchronisation is based on the lock-step method. The chapter illustrates the SystemC simulation cycle and the developed SystemC-A simulation cycle. It also overviews various synchronisation methods from the literature such as Backplane, Ping-pong, and Calaveras.

Chapter 6 presents four case studies to verify the functionality of SystemC-A mixed-signal simulator. The first two are simple case studies, the Van der pol oscillator and the Lorenz chaos. The other two are complex electrical case studies, a Switched Mode Power Supply (SMPS) and a 2GHz Phase Locked Loop (PLL)-based frequency multiplier, where new noise simulation methods are developed to simulate the PLL.

In Chapter 7, an electromagnetic case study of nonlinear ferromagnetic hysteresis based on Jiles-Atherton (JA) model is modelled and simulated as a challenge to SystemC-A to model non-electrical system. Due to the difficulties and instability of the current JA model, a new model is developed based on a timeless discretisation technique to integrate the magnetisation slope.

A mixed-domain system of automotive vibration isolation is modelled and simulated in Chapter 8. The system is a nonlinear complicated mixed electrical-mechanical-hydraulic domains and of excellent challenge for validating SystemC-A functionalities. The system was regulated using a suite of three complex controllers.

Finally, Chapter 9 summarises the presented work and the main achievements are highlighted and discussed. A number of concluding remarks are drawn. Further aspects which are not covered in this research are highlighted and could provide a basis for future work.

Chapter 2

Literature Review

Chapter 1 highlighted the importance and challenges of modelling AMS systems, suggesting SystemC as C++ based digital simulator to be linked to the proposed AMS extension to form SystemC-A mixed-signal simulation environment. Hence, before explaining the work, this chapter is to explain the following: Section 2.1 illustrates SystemC, its definition, architecture and models of computation. Concepts behind the idea of enabling a software language to model hardware are explained in Section 2.2. In Section 2.3 a literature review on modelling AMS systems is presented and serves as an orientation for this research. In Section 2.4, VHDL-AMS is demonstrated as the most popular and widely used HDL. It's constructs and approaches in modeling AMS systems are playing a reference to the development of SystemC-A.

2.1 SystemC

In September 1999 the Open SystemC Initiative (OSCI) [10] was announced at the Embedded Systems Conference in San Jose, California. At the same time,

the first version of SystemC V0.9 was announced and made available for free web download. OSCI is a growing community of leading electronics companies comprising EDA vendors, IP providers, semiconductor manufacturers, system design, and embedded software companies. In addition, OSCI includes universities and individuals dedicated to supporting and advancing SystemC as an open source standard to provide a common C++ modelling style for the entire electronics industry. To date, more than 37,684 registered licensees have downloaded SystemC from the OSCI website at www.systemc.org.

The first version of SystemC V0.9 was the result of a technical cooperation between Synopsys, Coware, and Frontier Design who all are leading EDA vendors [34]. This collaboration is the first of its kind in the EDA industry and promises to quickly establish a de-facto modelling standard.

In March 2000, SystemC V1.0 was released. That version of the language was limited to the behavioural and RTL modelling and it lacked many system level modelling features. Then, SystemC V2.0 was released in October 2001. It contains many system level modelling features which include channels, interfaces and events [44]. The latest version of SystemC V2.0.1 was released in May 2003. It is reportedly a bug free version with major communications and interface enhancements. OSCI submitted the SystemC LRM [10] to the IEEE for standardisation after an initial period of rapid adoption and evolution. On December 12, 2005, the IEEE approved the SystemC IEEE 1666 standard. A working group has been established in 2003 [45] aiming to extend SystemC to modelling AMS systems, but this aim has not been accomplished yet.

2.1.1 Language Definition

SystemC [10] is defined as a C++ class library and methodology that can be used to effectively create a cycle-accurate model of software algorithms, hardware architecture and build interfaces of SoC and system level designs. Using SystemC, a designer can create a system level model, quickly simulate to validate and optimise the design, explore various algorithms, and then end up with an executable specification of the system.

SystemC class library contains the necessary constructs to model system architecture, including hardware timing, concurrency, and reactive behaviour, that are missing in standard C++ [18]. C++ is an object-oriented programming language which has the ability to extend the language through classes, without adding new syntactic constructs.

OSCI released several documents which define the SystemC syntax and semantics, namely the User Guide [21], Functional Specification [46] and lately the Langauge Reference Manual (LRM) [10]. These documents are included in the SystemC installation package.

2.1.2 Design Flow

Figure 2.1 illustrates a typical simulation methodology in SystemC environment [47]. The designer writes SystemC models at the system level, behavioural level, or RT level augmented by the SystemC class library. The class library consists of a set of header files describing the implementation of hardware objects such as concurrent and hierarchical modules, ports, and clocks. The class library also contains a lightweight kernel for scheduling the processes. The header file (systemc.h) represents the class library and must be included in the user's code. The user's SystemC code and testbench can be compiled and linked together with the

class library using any standard ANSI C++ compliant compiler. The compilation result is an executable model of the design. Additionally, trace files can also be generated to view the history of selected signals using a standard waveform display tool.

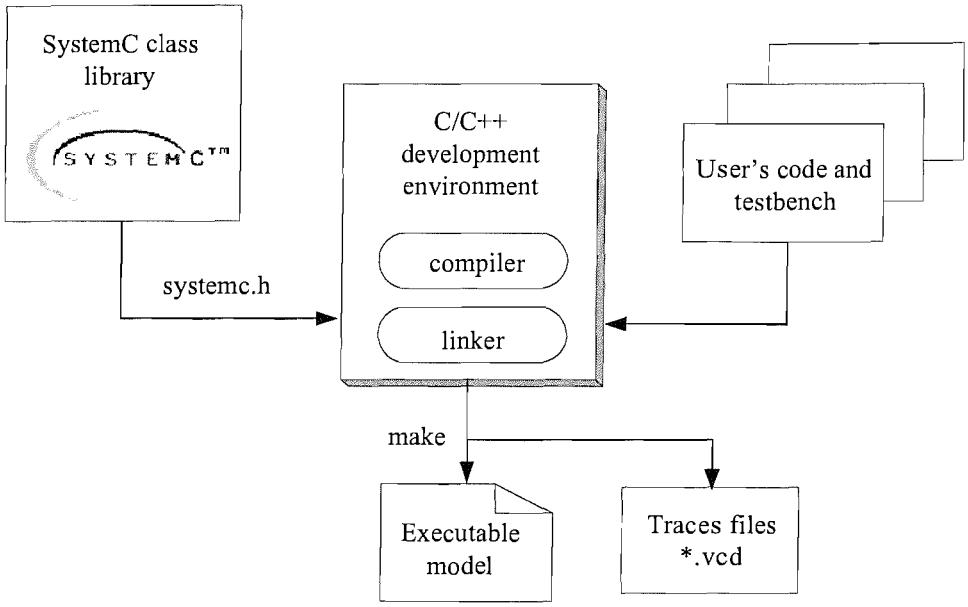

FIGURE 2.1: SystemC design flow (see Section 2.1.2).

2.1.3 Language Architecture

Figure 2.2 summarises the SystemC language architecture [48]. In the figure, the C++ standard forms the basis of SystemC, while the heart of SystemC is its simulation kernel. Then, there are all other essential features of an HDL including modules, processes, ports, signals, a rich set of data types, clocks, cycle-based simulation, multiple abstraction levels, debugging support, waveform tracing and communication protocols. The following paragraphs explain some of the most important constructs of SystemC.

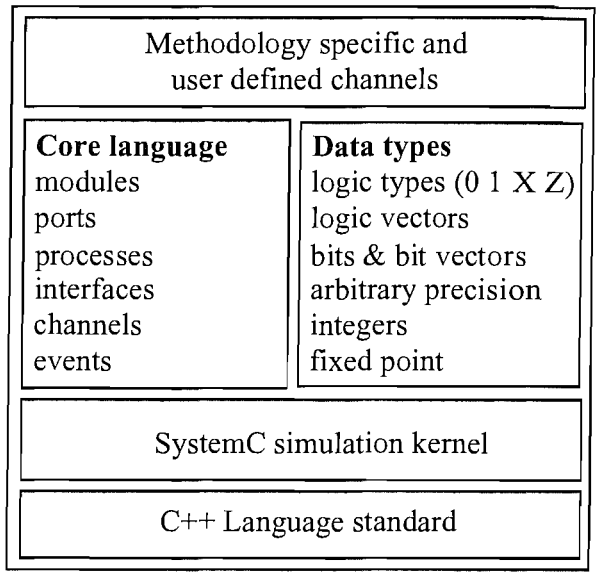

FIGURE 2.2: SystemC language architecture.

Modules

In SystemC the structural decomposition is specified with the SC_MODULE macro. Modules are the basic building blocks of any SystemC model. They can be hierarchical, containing instances of other modules. Modules contain processes which are the basic units of concurrent activity within SystemC.

Processes

In VHDL [8], concurrent behaviour is modelled using *processes*. In Verilog [9], concurrent behaviour is modelled using *always* blocks. In SystemC, concurrent behaviour is also modelled using *processes*. Processes are used to describe functionality and have an associated sensitivity list, a list of signals that trigger the execution of the process. SystemC provides three types of processes:

- SC_METHOD: It behaves like a function call and executes its body from the beginning to the end every time it is invoked. SC_METHOD offers the best simulation performance since it does not have its own thread of execution and hence cannot be suspended.

- SC_THREAD: It has its own thread of execution. A thread can be suspended at any time point and resumed at that point the next time it is entered. The performance is usually somewhat slower than that of an SC_MODULE due to context-switching overhead.
- SC_CTHREAD: A clocked thread has its own thread of execution and can be suspended and resumed at any point. A clocked thread is completely synchronised to a clock and is triggered by a transition on either positive edge or negative edge of a clock signal.

Clocks

SystemC provides a notation for clocks as special signals. Multiple clocks with arbitrary phase differences are supported. For example a 1μ second clock can be declared as:

```
sc_clock clk1 ("clk1", 1, SC_US)
```

The *sensitive*, *sensitive_pos*, *sensitive_neg* keywords can be used to synchronise a process with a clock.

Ports, Channels and Interfaces

The necessary elements of process communication in SystemC are ports, channels and interfaces. Modules are connected to each other via ports. SystemC supports single directional and bi-directional ports (*sc_in*, *sc_out*, *sc_inout*). Channels create connections between module ports allowing modules to communicate. Channels in SystemC can be either primitive or hierarchical. Primitive channels do not exhibit any visible structure, do not contain processes and cannot directly access other primitive channels. Examples of such channels are *sc_signal* (classical signals),

sc_mutex (used to model mutual exclusion) and *sc_fifo* (used to model queues). Hierarchical channels, on the other hand, are modules, which means that they can have structure, can contain other modules and processes and they can directly access other channels.

Ports are connected to channels through interfaces, where interfaces define sets of methods channels must implement. Ports are objects through which modules and hence processes can access a channel interface. Figure 2.3 [49] shows a simple design with three modules connected to a channel.

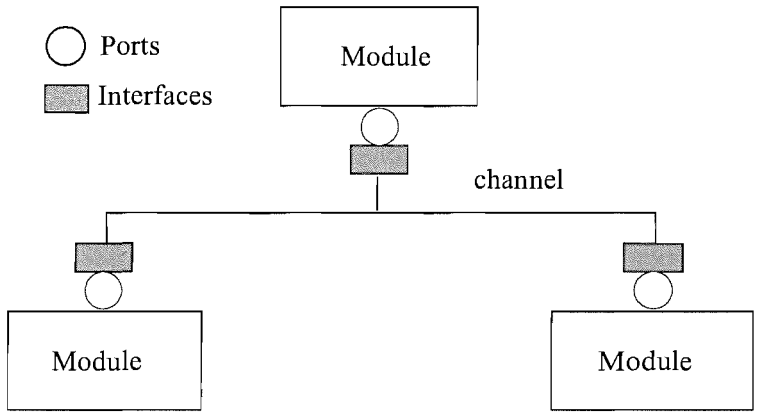

FIGURE 2.3: An illustration of ports, channels and interfaces of SystemC (see Section 2.1.3).

An example which illustrates a SystemC module is shown in Listing 2.1 of a D-flip flop with asynchronous reset [21]. The example illustrates the module's connectivity, declarations and process.

2.1.4 Data Types

In addition to the standard C++ data types, SystemC has a rich set of data types to support multiple hardware design domains and abstraction levels [21]. This is different to many other HDLs, such as Verilog, that only support bit and bit-vectors as signal types. The fixed precision data types (*sc_fixed*, *sc_ufixed*, *sc_fix*, *sc_ufix*) allow for fast simulation and are not found in other HDLs. The arbitrary

precision types (*sc_bigint*, *sc_biguint*, *sc_bv*, *sc_lv*) can be used for computations with large numbers whereas the fixed point data types can be used for DSP applications. There are no size limitations for arbitrary precision SystemC types. SystemC supports two-valued (*sc_bit*) and four-valued (*sc_logic*) logic data types.

```
1  #include "systemc.h"//SystemC class library
2
3  SC_MODULE(dffa) {
4      sc_in<bool> clock;//input and output ports
5      sc_in<bool> reset;
6      sc_in<bool> din;
7      sc_out<bool> dout;
8
9      void do_ffa(){// process body
10         if (reset){
11             dout = false;
12         }
13         else if (clock.event()){
14             dout = din;
15         }
16     };
17
18     SC_CTOR(dffa) {// constructor of module
19         SC_METHOD(do_ffa);//a process
20         sensitive(reset);// sensitivity list
21         sensitive_pos(clock);
22     }
23 };
```

LISTING 2.1: SystemC example: D flip flop with asynchronous reset.

2.1.5 Simulation Kernel

The appropriate timing behaviour of an executable SystemC model is controlled by the simulation kernel [10]. The simulation kernel is a lightweight cycle-based scheduler that allows high speed simulations. It is responsible for activating processes according to their sensitivity list, executing statements within different processes concurrently, scheduling assignments to signals and propagating their changed values through the whole system hierarchically.

The SystemC scheduler has support for both software and hardware-oriented modelling. Similar to VHDL and Verilog, the SystemC scheduler supports delta cycles. A delta cycle consists of separate evaluate and update phases, and multiple delta

cycles may occur at a particular simulated time. Delta cycles are useful for modelling fully-distributed, time synchronised computation as found for example in RTL hardware. In SystemC, using *notify()* with a zero time argument causes the event to be notified in the evaluate phase of the next delta cycle, while a call to *request_update()* causes the *update()* method to be called in the update phase of the current delta cycle. Using these facilities, channels which model the behaviour of hardware signals can be constructed. The SystemC simulation algorithm is shown in Algorithm 2.1 [10].

Algorithm 2.1: SystemC simulation algorithm.

1: Initialisation Phase: Execute all processes (except SC_CTHREADs) in an unspecified order.
2: Evaluate Phase (EPh): Select a process that is ready to run and resume its execution. This may cause immediate event notifications to occur, which may result in additional processes being made ready to run in this same phase.
3: If there are still processes ready to run, go to EPh.
4: Update Phase: Execute any pending calls to update() resulting from request_update() calls made in EPh.
5: If there are pending delayed notifications, determine which processes are ready to run due to the delayed notifications and go to EPh.
6: If there are no more timed notifications, simulation is finished.
7: Advance the current simulation time to the earliest pending timed notification.
8: Determine which processes are ready to run due to the events that have pending notifications at the current time. Go to EPh.

2.1.6 Models of Computation

A Model of Computation (MoC) [48] can be defined by, the model of time employed (real-valued, integer-valued, untimed), the event ordering constraints within the system (globally ordered, partially ordered) or the rules for process activation.

In SystemC, the simple and flexible synchronisation capabilities provided by events and *wait()* call allow a broad range of different channel types to be implemented without having to change the underlying simulation kernel [48]. All the required

functionality to create a specific MoC is already presented in the simulation kernel. Thus, SystemC supports very powerful generic MoCs [50]. While the global model of time is fixed to integer model, designers can construct specific channels to achieve their precise rules for communication between processes, process activation and system wide event ordering. Although continuous time modelling still cannot be constructed in SystemC, virtually any discrete time system can be modelled in SystemC.

The most important MoC is the Transaction Level Modelling (TLM) [51]. TLM is a high level approach to modelling digital systems where the details of inter-module communication are separated from the implementation details. Transactions requests are made by calling the interface functions of the channel models that encapsulate the low level details of the information exchange. Transaction's interface thus focuses on the functionality of the data transfer rather than its implementation. Many researchers have demonstrated the SystemC capability to model systems at TLM [33, 51, 52] and indicated that using SystemC at TLM provides a gain in simulation speed.

2.2 Modelling Hardware in C/C++

In the recent past, a few projects have been looking into methods of using C/C++ as an input to current design flows [2]. Modelling hardware in C/C++, requires the following list of HDL features [32].

- Concurrency : Hardware is inherently parallel, while C/C++ models are inherently sequential. Therefore, a notion of processes should be introduced to encapsulate programs and execute them concurrently. Then a system is described as a network of processes.

- Signals : Hardware processes need to be connected to signals or channels to communicate with each other.

- Reactivity : Hardware systems are reactive and in continuous interaction with their environment. Hence, reactivity is implemented in C++ through event-driven approach.

- Data abstraction : In addition to the data types supported by C++ for sequential programming, hardware modelling needs arbitrary precision signed and unsigned integers, bit vectors and fixed point types.

Many C/C++ and C-based HDLs have been developed and used for modelling and synthesis. Examples of C-based HDLs are, HardwareC [53] from Stanford University and CONES [54] from AT&T Bell Laboratories. Examples of C/C++ based languages other than SystemC are Esterel C [55] from Cadence, Handle-C [19] from Oxford University (now marketed by Celoxica), BachC [56] from Sharp Laboratories, SpecC [20] from the University of California, Ocapi from IMEC [57], SystemC++ from C Level Design Inc. and Cynlib from CynApps [58]. Some other flavours of C/C++ based languages have also been developed and used internally throughout the industry as illustrated in Table 2.1.

Language	Founder	Based on
HardwareC	Stanford University	C
CONES	AT&T	C
SystemC	Synopsys	C/C++
Esterel C	Cadence	C/C++
Handel-C	Oxford University	C/C++
BachC	Sharp Laboratories	C/C++
SpecC	University of California, Irvine	SpecChart, C/C++
OCAPI	IMEC	C/C++
SystemC++	C Level Design	C/C++
CynLib	CynApps	C/C++

TABLE 2.1: C/C++ based design languages.

2.3 Mixed-Signal Modelling With SystemC

While using SystemC in modelling, verification, and synthesis of digital systems is well established and proven (Appendix B gives a literature survey on SystemC applications), the concept of extending SystemC to AMS systems design is novel. Recently, a number of research results have been presented suggesting different frameworks and methodologies. The AMS frameworks are based on different C++ classes, methods and libraries, some are dedicated to specific applications, some seem awkward to use. Each proposal has its strengths and weaknesses which will be discussed after giving overviews of them.

For instance, O'Nils et al [59] presented a methodology for quantification of noise coupling in mixed-signal systems called BeNoC. The presented method facilitates seamless quantification of both the power distribution network and substrate noise coupling at behavioural level. Starting from a behavioural model of the system captured in SystemC, wrappers are added to each block in the behavioural model. These wrappers add an estimated power consumption model for each block, which is triggered by events. The noise coupling simulation is then done by connecting different blocks according to a virtual layout and technology parameters. The resulting noisy substrate or noisy power distribution network can then be fed back into the behavioural level. Thus, effects on the system behaviour can be analysed. The simulation results compared with circuit simulations in SPICE showed that their approach is two orders of magnitude faster than SPICE. Also, the simulation can be done long before the circuits have been designed.

An AMS simulation framework is presented by Bonnerud et al [60] for simulation of Analogue to Digital data Converters (ADC). The framework contains a C++ mixed-signal module library that includes a set of flexible and customisable primitive, compound modules and testbenches. The primitives implement functional models of AMS basic building blocks such as flash ADC, switched capacitor

Digital to Analogue Converter (DAC), operational amplifier, track-and-hold amplifier, and analogue adder/subtractor. More application specific primitives are also included such as a switched capacitor multiplying DAC. Furthermore, they have implemented a clocking scheme for scheduling the analogue and mixed-signal blocks called virtual clock to avoid multiple executions of these blocks due to the SystemC kernel. They have illustrated the usability of the framework by applying it to two case studies of pipelined ADCs with background calibration, achieving comparable accuracy to that of MATLAB.

Another AMS framework was presented by Conti et al [61] which allows the designer to describe analogue systems either at low or high level using analogue macromodels. Their time step scheme is based on the calculation of each analogue process to its adaptive time step and passing it to the blocks to which the process outputs are connected. They suggested a new implementation of an analogue block. It is a module composed of two kinds of threads, the calculus thread and activation threads, one for each input module. Each activation thread starts when a change occurs on the signals coming from the corresponding input module. The calculus thread that updates the state and output is activated only by signals coming from the activation thread, and then sending them to the connected analogue blocks. They have used two AMS examples to validate their framework, an oscillator made up of inverter chain and a complex mixed-signal fuzzy controller. The results were compared to other tools such as SPICE and Spectre achieving excellent results. The CPU time required for the 800n second oscillator simulation is 4.9 seconds for SystemC and 15.8 seconds for WinSpice. For the second example, the CPU time required for Spectre transient simulation is 4 hours and 38 minutes, only 3.3 seconds are necessary for a SystemC simulation with a factor of 5000 due to the very high level description used in SystemC. The framework is later called SystemC-WMS [62], where WMS stands for Wave Mixed Signal and energy wave signal is the analogue signal transmitted between analogue blocks.

Grimm et al [63] presented a top-down modelling and simulation methodology based on a refinement process by implementing a library called ASC. The design methodology refines an executable specification to concrete AMS architecture through three levels of refinement, executable, computation accurate model and pin accurate model. The ASC library provides an analogue or signal processing process type. The execution of the analogue processes is not controlled by the discrete event kernel; however the execution is controlled by a coordinator interface. Via this interface, a coordinator can call the signal processing C++ method by remote method invocation. For complex analogue processes, they used an external analogue simulator such as SABER or SPICE. By using multiple inheritances, analogue signals realise the interface used in discrete-event simulation. The signal class can be read or written by a discrete-event process without the need for an explicit converter as needed for example in VHDL-AMS for conversion between quantities and signals. The methodology was evaluated by designing a PWM controller.

Another design methodology presented by Romberg and Grimm [64] based on refinement process. It started with a system specification captured by a new graphical design notations called HyCharts, and then translate it to SystemC with the AMS library ASC in [63]. The authors claimed that starting with Hycharts which based on formal semantics is precisely to capture the continuous/discrete behaviour.

Einwich et al [65] presented a framework to support signal processing dominated application. The framework is based on analogue extension for linear DAEs and frequency domain simulation. The linear DAE solvers are integrated into the synchronous data flow design. The analogue extension includes a library of electrical circuit components and transfer functions. The synchronisation between the synchronous dataflow and linear continuous time is using a fixed time step

and kept as simple as possible. Before the first delta cycle of a time step is executed, all analogue simulators are executed, reading the old discrete values and producing updated output signals which are then used by the digital processes. The main characteristics of their framework are: no overall analogue equation system, no iterations are used to solve the system, no solvability problem for the overall system, defining analogue module (*sca_module*), ports (*sca_port*), channels (*sca_channel*) and interfaces (*sca_interface*). The concept was illustrated with the design of a telecommunication system consisting of a subscriber line interface and a Codec filter system, including digital hardware and software and an analogue filter. Simulations were about 20 times faster than those of SABER. A simulation of a 40m seconds time interval took 122 seconds in SystemC, while on SABER - 2679 seconds. The framework is later called SystemC-AMS [66].

SystemC-AMS [66] is used by Markert et al [67] to model inertial navigation system. The system consists of analogue sensors and digital coordinate transformation part together with a PC-based software part. The analogue sensors measure the signal from the environment and converts them to digital values. The system was originally modelled in VHDL-AMS in more details and SystemC-AMS offered the designers high level insight of the system.

A framework called AnalogSL is presented by Grimm et al [68] for the creation of behavioural models of analogue power drivers. The main objective of their framework was to achieve a speed up over other simulators. AnalogSL provides classes of components such as resistors, capacitors, coils and transistors which can be instantiated to form a netlist and then simulated by a very fast and efficient algorithm for linear DAE. Analogue power drivers are usually designed bottom-up, from a netlist to behavioural level. The coupling of the analogue behavioural model with discrete-event simulators corresponds to the coupling of different processes in discrete simulators. When an input value has changed, the method checks

whether the dominant cycles have changed, otherwise, it calculates the actual internal states and sets up new equations.

A mixed-signal SystemC design environment has recently been proposed [40] for behavioural modelling, simulation, and performance evaluation of microelectromechanical and microelectrofluidic SoCs. Composite microsystems combine microstructures with solid-state electronics to integrate multiple coupled-energy domains, e.g., electrical, mechanical, thermal, fluidic, and optical, on a SoC. Continuous-flow systems microelectrofluidic such as microvalves, micropumps and channels are modelled by DAEs and PDAEs (partial DAEs) in SystemC. The continuous equations are solved by using the regular function procedures or process, and code various DAEs solvers with SystemC, such as derivative and integral, and add them into a SystemC component behaviour model. Moreover, besides the original simulation clock, they implemented a higher frequency clock to provide a series of time intervals for more accurate DAEs function solutions. They have used the relaxation based numerical integration techniques coded in SystemC to solve these DAEs.

A method to link SystemC digital modules with Verilog-AMS has been proposed by Birrer and Hartong [69] in order to simulate AMS of analogue/RF systems. It was achieved by automatic flow envelops SystemC modules in Verilog-AMS warpers, where they can be used in a schematic flow and treated just like standard Verilog modules.

The surveyed frameworks above presented new ideas and methodologies, however, none of them introduced a general environment to model and simulate AMS and mixed-domain with different abstraction levels for generic applications. Most of the frameworks are application specific extensions [59, 60, 65, 66, 63], or abstraction level specific [59, 60, 61, 68]. Some introduced tedious methods for timing [61, 63], others introduced non popular methods to capture models [64], others

cannot handle complex analogue processes [63, 68, 66], and others bringing all the details of modelling to the user level [61, 66]. Nearly all the frameworks were validated using examples with small analogue parts and a big digital part.

A study group was established in 2003 [45] following a proposal submitted to the SystemC board of directors to form an OSCI working group to develop AMS extensions to SystemC. The founders presented and discussed the foundations on which the analogue and mixed-signal extensions of SystemC, named SystemC-AMS, will be based [70, 71, 45, 72, 66]. They set out a plan for a 3-phase development, each phase adding new capabilities as follows:

1. Support for signal processing dominated applications. This includes:
    - Linear dynamic continuous-time, including transient, small-signal AC and noise simulation. Time-domain simulation with a fixed time step.
    - Predefined linear operators (Laplace transfer function, zero-pole transfer function, state-space equations).
    - Linear network elements (electrical element library: R, L, C, sources).
    - Continuous behaviour encapsulated in static dataflow modules.
    - Synchronisation between discrete event and continuous time MoCs using static dataflow semantics.
2. Support for RF/wireless applications. This includes:
    - Support of non linear DAEs and their simulation using variable time steps.
    - Formulation of implicit equations, e.g. true simultaneous statements.
    - Frequency-domain simulation.
    - A mixed-signal library with more complex functional (signal-flow) models, e.g. amplifiers, converters.

3. Support for automotive applications. This includes:

    - Specialised continuous-time MoCs, e.g. for power electronics or mechanical systems.
    - Support of network law models.
    - Add network-law mixed-domain models to the mixed-signal library.
    - Definition of a generic synchronisation mechanism between discrete-time and continuous-time MoCs, including software MoCs.

Although the SystemC-AMS study group was established in 2003, the development is still in the initial stages. As the group reported in September 2005 [66], SystemC-AMS currently lacks the ability to setup equation systems and an analogue equation solver is still under development.

SystemC users are waiting for AMS to SystemC to design and verify their entire system in one environment [40, 73]. For instance, Cuenin et al [73] designed an AMS IP which represents the external communications between a SoC and its environment. The AMS IP model is written in SystemC in combination with VHDL-AMS description of the analogue blocks.

2.4 VHDL-AMS

Today's HDLs are classified into three categories, digital, analogue, and AMS HDLs. Examples of digital HDLs are VHDL and Verilog, they are based on event-driven techniques and a discrete mode of time. Analogue HDLs support the description of systems of DAEs. Examples of analogue HDLs are SpectreHDL from Cadence and Verilog-A from the Open Verilog International. Analogue HDLs support network semantics and behavioural descriptions. AMS HDLs support both

event-driven techniques and DAEs. The most popular AMS HDLs are VHDL-AMS and Verilog-AMS. As the names imply, they are extensions to the classical Verilog and VHDL digital HDLs. Though, these languages have different strengths and weaknesses, they are intended to be used for the same types of circuits, in the same ways, to produce similar results. This section overviews VHDL-AMS one of these HDLs. VHDL-AMS constructs and implementations serve as inspirations to SystemC-A development in order to follow methods familiar to the design community.

VHDL-AMS [14] is one of the major mixed-signal HDLs on today's CAD tool market. It is a superset of the IEEE standard 1076-1993, with AMS extensions. The new, complete language is defined by the IEEE standard 1076.1-1999. The standardisation by the IEEE means that the language can be used by different tool vendors. The language definition does not specify the whole internal operation of the simulator in the analogue domain, leaving the details of implementation for the tool vendors. Many companies have been developing solutions for the implementation of the language standard into their own simulation packages. For example, SystemVision from Mentor Graphics, Simplorer from ANSOFT and INCISIVE from Cadence Designs. There are several public domain VHDL-AMS parsers available for free use [74, 75, 76, 43] and also there are some validation examples published on the web [43].

VHDL-AMS aims to provide a language to simulate a variety of physical domains that model complex systems, such as mechanical [77], chemical, automotive [78], and mechatronic [79] systems. VHDL-AMS has also been used to model Micro-Electro-Mechanical Systems (MEMS) [80], and in the simulation of self organising neural systems [81]. VHDL-AMS allows different parts of the system to be described at behavioural level [82] and component level of abstraction [83].

Currently, most digital systems can successfully be synthesised using VHDL. It is

hoped that VHDL-AMS will also provide a basis for a new approach to AMS circuit synthesis. For instance, a synthesis system for high frequency analogue filters from VHDL-AMS [84] has already been developed. A mixed-signal VHDL-AMS based synthesis system from behavioural models has also been proposed [85].

VHDL-AMS new language elements facilitate writing analogue models as a set of mathematical equations describing the behaviour of the model. Examples of language elements are: *Simultaneous equation*, *Quantity*, *Terminal*, *Nature* and *Tolerance* [86]. A typical VHDL-AMS model consists of an entity and one or more architectures. The entity specifies the interface of the model to its environment. It includes types of model's ports and the definition of its generic parameters. The architecture contains the implementation of the model in structural or behavioural descriptions or both. The following sections summarise some of VHDL-AMS features extracted mainly from [86], [87], and [88].

2.4.1 Quantities

Quantity objects represent the unknowns in DAEs. They can be scalar or composite (arrays and records). Quantities can be declared anywhere a signal can be declared and it may appear in expressions, interfaces and simultaneous equations. Quantities have several forms, it can be a free quantity, or an interface quantity in a port list of a model to support signal flow modelling. It has two modes *in* and *out* specifying the direction of signal flow. For instance, Listing 2.2 shows the entity declaration of a signal flow model. A port declaration can be quantity, signal, or terminal declarations.

```
1 entity example is
2         port (quantity input1, input2: in REAL;--scalar quantity
3               quantity A, B, C: out REAL_vector(2 downto 0));--array quantity
4 end entity example;
```

LISTING 2.2: VHDL-AMS entity declaration of a signal flow model.

Other forms of a quantity are branch quantity and source quantity, they will be explained in Section 2.4.3 and Section 2.4.7 respectively. Quantities have a number of predefined attributes, e.g. *Q'Dot* is a quantity holding the derivative of quantity Q with respect to time. Other attributes are Laplace transfer function (*Q'ltf(num,den)*), time integral (*Q'integ*) and time delay (*Q'delayed(t)*). For a model to be solvable, the number of simultaneous equations defined should equal to the total sum number of through quantities plus free quantities plus interface quantities with mode out.

2.4.2 Simultaneous Statements

Simultaneous statements are used to describe DAEs. They are analogous to concurrent signal assignment in digital models. There are several simultaneous statements, the *simple simultaneous statement* which is an expression usually denoting an equation. *Simultaneous case statement* chooses different simple simultaneous statements, depending on the condition of the case statement. *Simultaneous if statement* chooses different simple simultaneous statements, depending on the condition of the if statement. All simultaneous statements are concurrent, the order in which they appear in the model is not important for the calculation of the result. Another form is simultaneous *procedural* statement to handle statements sequentially. Simultaneous statements may contain and refer to signals, quantities, constants, literals, and functions.

An example of a VHDL-AMS model of a signal flow amplifier is shown in Listing 2.3 [88]. The example illustrates the entity-architecture pair of a VHDL-AMS model with the new language constructs.

```
1   entity amplifier is
2       generic (gain: REAL :=REAL'High);-- default infinity gain
3       port (quantity input: in REAL;
4             quantity output: out REAL);
5   end entity amplifier;
6
7   architecture ampl of amplifier is
8   begin
9       if gain = REAL'High use -- Simultaneous if statement
10          input == 0.0;
11      else
12          output == gain * input;
13      end use;
14  end architecture ampl;
```

LISTING 2.3: VHDL-AMS model of a signal flow amplifier.

2.4.3 Provision for Network Topology

Analogue systems presented in the form of networks, for example, electrical circuit obeying Kirchhoff's Laws, can be described in VHDL-AMS using *Branch* quantities, namely *across* and *through* quantities. *Across* quantities represent effort-like effects such as voltage, temperature, or pressure. *Through* quantities represent flow-like effects such as current, heat flow rate, or fluid flow rate. For instance, a resistor is governing by ohm's law which relates the voltage across (the *across* quantity) and the current through the resistor (the *through* quantity): $i = \frac{v}{r}$.

A branch quantity must be declared with reference to two terminals. A *terminal* is a fixed point in the structure of a physical model, e.g. an electrical node. It declared to be of some physical discipline (or nature), i.e electrical, thermal, fluidic, etc. *Nature* definition includes the types of *across* and *through* quantities incident to a *terminal* of the specified domain, and the common reference terminal shared by all terminals (e.g. electrical ground).

The following statements declare two terminals *t1* and *t2* of nature *electrical*, an *across* quantity *v*, and a *through* quantity *i* between the terminals.

```
terminal t1, t2: electrical;
quantity v across i through t1 to t2;
```

The *across* quantity represents the potential difference between the terminals and the *through* quantity represent current-carrying branch. The data type of a branch quantity is derived from the nature of its terminals. In the example above the *across* quantity v is of type *voltage*, and the type of the *through* quantity i is *current*.

2.4.4 Tolerances

VHDL-AMS defines the concept of a *tolerance group* where each quantity belongs to such a group. This is to allow a user to control how close to zero the solution of the DAEs must be. The tolerance group of a quantity is defined in the *subtype* of the quantity or, at the quantity declaration or, in the nature declaration for a terminal. For example, voltage and current subtypes,

```
subtype voltage is REAL tolerance "default_voltage";
subtype current is REAL tolerance "default_current";
```

where "default_voltage" and "default_current" define the tolerance group of the subtypes. The simultaneous statements can also have a tolerance, it will override any declared tolerances. For example,

```
X==Y'dot tolerance "low_voltage";
```

However, the default tolerance group of a simple simultaneous statement is the tolerance group of its quantities. VHDL-AMS does not define how tolerances are used. It is the responsibility of the tool vendor to define how the tolerance characteristics are calculated.

2.4.5 Analogue/Digital (A/D) Interaction

A/D interface is implemented by quantity attribute *Q'above(E)*. During a simulation, when the value of a quantity Q crosses a threshold E, an event occurs on the Boolean signal *Q'above(E)*. The value of *Q'above(E)* is *TRUE* if $Q > E$ and *FALSE* if $Q < E$. There is also a hysteresis in the signal change. The result of *Q'above(E)* stays *FALSE* as long as Q has not reached the upper boundary of E, which is $E+\delta$. *Q'above(E)* stays *TRUE* as long as Q is bigger than the lower boundary of E, which is $E-\delta$. The size of δ is implementation-dependent.

An event is generated at the instant of the threshold crossing, deriving other digital parts, because the result of *Q'above(E)* is a digital boolean signal. This mechanism of threshold crossing can also be used for A/D conversions in system model. For example, the following statement implements the behavior of an ideal comparator.

```
S <= '1' when Q'above (0.0) else '0'
```

where Q is a quantity and S is a signal. *Q'above(E)* can also be used in the sensitivity list of a process or a wait statement to trigger the process based on an analogue signal.

2.4.6 Digital/Analogue (D/A) Interaction

D/A interface also needs a mechanism of translation of data values. When a digital signal appears in a simultaneous statement, it introduces a discontinuity in the DAEs solver because of its discontinuous nature. VHDL-AMS defines a *break* statement to complement the simultaneous statement and notify the analogue solver exactly about when the discontinuity occurs. The analogue solver responds by re-initialisation at the exact time where the break occurred. However, it seems

it is a difficult task for tool vendor to implement the *break* statement. The break statement is not implemented yet in any recent VHDL-AMS simulator.

An alternative mechanism is to use one of the predefined quantity attributes, *S'Ramp(trise, tfall)* or *S'Slew(rising slope, falling slope)*, where S is a signal of a floating point type. *S'Ramp* ramps linearly over the specified rise and fall time from the previous value of S to its new value, starting at the time of the event. *S'Slew* does the same, but with specified slopes.

2.4.7 Small-Signal Frequency Domain and Noise Modelling

VHDL-AMS provides the *source* quantity, to support small-signal frequency-domain (AC) and noise simulations. Source quantities provide stimulus for frequency domain simulation. There are two types, *spectral source* quantity and *noise source* quantity. Spectral source quantity allows the modeller to specify a magnitude M and phase ϕ of a stimulus in the form of a sin wave ($v(t) = M\cos(\omega t + \phi)$). The following statement gives an example of a spectral source quantity definition:

```
quantity ac:real spectrum magnitude, math_2_pi*phase/360.0;
```

During time-domain simulations, source quantities have a value of a zero. Noise source quantities are used in a similar manner, as illustrated bellow:

```
quantity thermal_noise:real noise 4.0*k_boltzmann*temp*res;
```

The expression following the reserved word *noise* specifies the noise power which is not required to be static. The noise source quantity is used in the model by simply adding *thermal_noise* quantity to a simple simultaneous statement. E.g. to model the thermal noise in a resistor:

```
v==i*resistance+ thermal_noise;
```

2.5 Concluding Remarks

This chapter has presented several subjects, SystemC and its applications especially modelling AMS systems, concepts of modelling hardware in C/C++ and an overview of VHDL-AMS. A number of concluding remarks could be made. Firstly, through the surveyed literature in SystemC applications, it was apparent that SystemC is powerful in industrial applications. Also, there is a trend towards C++ based HDLs for many reasons such as raising the abstraction level and consequently gaining higher simulation speed and most importantly hardware/software co-design. However, the main focus was the ability of SystemC to be an environment of hardware and software modelling at system level design. Furthermore, the chapter demonstrated some initial attempts towards AMS modelling using SystemC which showed the need for such ability. Also, it showed what have been accomplished so far in the field and where it is stopped.

Moreover, through the summary of VHDL-AMS as one of the existing tools and methodologies in the field of AMS, the chapter showed the strength and weakness of such a language. VHDL-AMS cannot cope with the recent need to co-model hardware and software for SoC applications. The developed AMS extension will closely follow the concepts developed in VHDL-AMS in particular the concepts explained in this chapter.

Chapter 3

AMS Modelling Syntax

This chapter describes new CAD methods and elements which are developed in the course of this research in order to model AMS systems and present a new language named SystemC-A. The new elements cover the most important aspects of AMS modelling. The AMS syntax has been designed to facilitate model development assuming minimal programming knowledge of a future user of SystemC-A. The style is similar to a SPICE-like net-list or VHDL-AMS simultaneous equation or interconnected blocks as in any HDL.

Modelling of an analogue system requires a set of Differential and Algebraic Equations (DAE) which should be easy to define, automatically built and updated, and then numerically solved. Therefore, methods and constructs should be defined to support the simulation of analogue systems represented by DAEs. The new language constructs will be proceeded by the prefix *sc_a_*, where *sc* denotes SystemC and *a* for analogue.

In Section 3.1 the chapter starts by introducing some of the C++ object-oriented concepts which are used in coding. Then, Section 3.2 introduces the first essential element necessary to support analogue systems, namely *analogue system variable*

whose objects represent the unknowns in the set of DAEs, such as circuit nodes and flow variables. The next extension presented in Section 3.3 is *component*, a new class which can be extended according to a particular physical domain or application. Its objects can be used to build general systems such as electronic circuits. A *component* class contains a virtual *build* method through which system equations are formed and solved at each time step. Section 3.4 deals with digital-analogue interaction issues, such as passing messages between the digital and analogue solvers and conversion between signals and system variables. Associated issues such as dealing with small time step sizes, implementing analogue stepping are also discussed. Section 3.5 gives a note about abstraction levels provided by SystemC-A. Finally, Section 3.6 concludes the chapter.

3.1 Preliminaries: Object-Oriented Programming

C++ [89] is an object-oriented language and based on the principle of structured programming. When a task is too complex to be described, it is broken down into a set of smaller component tasks (*classes*). A C++ class is a format for holding and interacting with data and has an interface called *constructor*. An object is a particular instance of a class. A program is a collection of constructed classes and a testbench. The testbench is a high level module in hierarchy which contains global signals, instances of classes, and provides stimulus to the system. The three defining properties of object-oriented programming are encapsulation, inheritance, and polymorphism. *Encapsulation* is to hide the details of a component into a class, to act as a fully encapsulated entity and used as a whole unit. Users need to know how to use it rather than how the class is working. Objects can only be accessed through their public interfaces, while the internal data and implementations remain hidden. This ensures that the code is safe from unwanted alterations.

C++ supports the idea of reuse through *inheritance*. A new class can be derived from an existing one and is called a *derived class* and thus inherits all its qualities but additional features can be added to it as needed. A hierarchy can be formed from derived classes. The existing class is called an *abstract base class* which may contain methods declared as *virtual* methods. When declaring a method of abstract base class to be virtual, the derived classes can override and redefine this method and act as a base method, this is called *polymorphism*. Therefore, polymorphism allows different classes to support the same interface but with different implementations. By utilising the fundamentals of object-oriented programming, a data structure called *linked-list* can be defined to store data dynamically. It consists of a chain of classes, which contains data and nodes. Nodes are pointers that stores the memory address of the following class keeping the list linked together. Linked-lists support four basic operations: insertion, deletion, traversing, and searching. A C++ class consists of *header* and *cpp* files. cpp files are the main body of a class. Header files are libraries of code implementing useful functions written by the user or by others. A user includes header files in the topmost of a cpp file and then the compiler will write the contents of the header files into the executable code of his program.

3.2 Analogue System Variables

In order to provide a mechanism for modelling non-linear AMS systems, the new language should provide a notation for DAEs. In the set of DAEs the analogue system variables are the unknowns.

The C++ concept of inheritance is used to define various types of analogue system variables, such as *node*, *flow*, and *free variables*. In SystemC-A, they represent a hierarchy of system variables, all derived from an abstract base class called

sc_a_system_variable as illustrated in Figure 3.1. Currently only three types of variables derived from the base class have been defined, and this proved enough to model the application examples presented later in this thesis (Chapter 6, 7, 8). The variable classes hierarchy can be extended further to model other types of applications. In a SystemC-A description, the total number of variable objects must correspond with the total number of analogue equations provided by SystemC-A component objects. This is to have a symmetric system matrix.

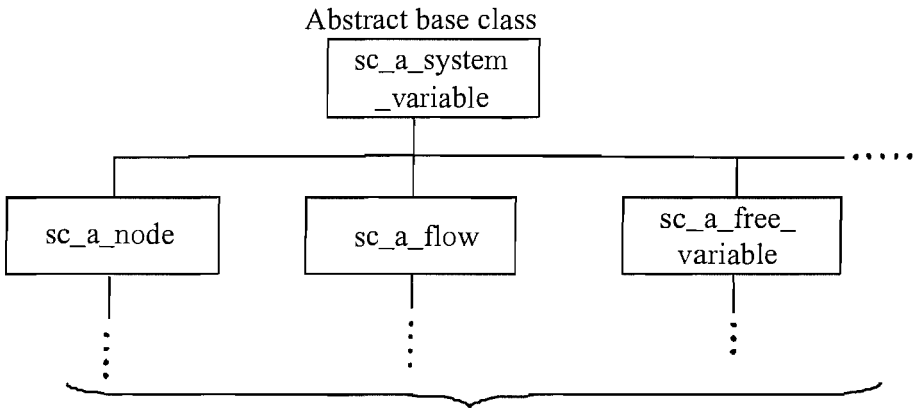

FIGURE 3.1: Analogue system variable inheritance hierarchy.

The base class constructor attaches each newly created system variable to a global linked-list as shown in Figure 3.2. The list will later be used by the analogue kernel to maintain the required connectivity between the system components and to build the underlying analogue equation set. This is done by scanning the linked-list of variables and assign integer numbers to them, and then use these numbers as indices to the system matrix to perform element searching and insertion functions. Accordingly, the order of the system will be known, where it is equal to the number of system variables.

System variables have a common method called *X()* to read a value of a particular system variable from system solution vector whenever required. The function's

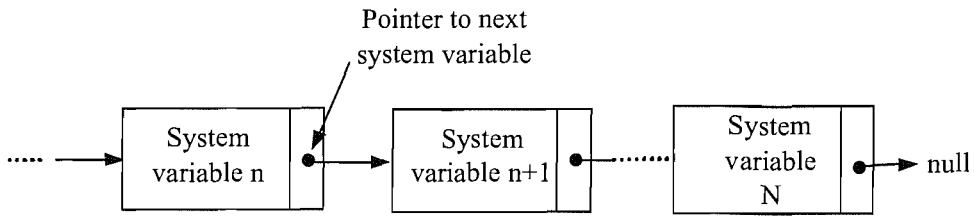

FIGURE 3.2: Linked-list of system variables.

counterpart in SystemC is *read()* to read a port or a signal. Also, SystemC-A supports *differentiator* (*Xdot()*) and *integrator* (*INTEG()*) operators to be performed on system variables. Their use can be demonstrated as follows:

```
A_dot=Xdot(n1);
B_intg=INTEG(n1);
```

where $n1$ is a system variable of any type. Integral and differential operators can be performed on non-SystemC-A variables by declaring a new SystemC-A variable ($n1Q$) and use the following version of the operators,

```
A_dot=Xdot(n1Q,n1);
B_intg=INTEG(n1Q,n1);
```

The following subsections describe the three different types of analogue system variables derived from the base class to support the examples in Chapters 6, 7 and 8.

3.2.1 sc_a_node

sc_a_node is used to represent a generic node. Objects of this class can be declared where a circuit would be defined. *sc_a_node* is instantiated in any SystemC module or SystemC-A component's constructor because it should be instantiated only once to maintain the correct order of the system. Objects of the *sc_a_node* class are instantiated as follows:

```
SC_MODULE(filter){//SystemC module
 ...
   sc_a_node  *n0, *n1, *n2;
 ...
   SC_CTOR(filter){//module constructor
   ...
      n2 = new sc_a_node("n2");
      n0 = new sc_a_node("0");
      n1 = new sc_a_node("n1");
   ...
      }
};
```

A special node is the reference node or ground. The ground node does not have a particular syntax in SystemC-A. It is instantiated in the same way other nodes do but with node name of "0". When scanning the nodes, the analogue kernel can recognise the reference node from its name. Accordingly, the analogue kernel will not raise the order of the system when scanning the reference node.

As in other HDLs, different types of ports and quantities can be developed in libraries to distinguish different physical domains, e.g. *Nature* in VHDL-AMS. Electrical, mechanical, and magnetic nodes as shown in Figure 3.3, can be derived from class *sc_a_node* for creating accurate physically based models.

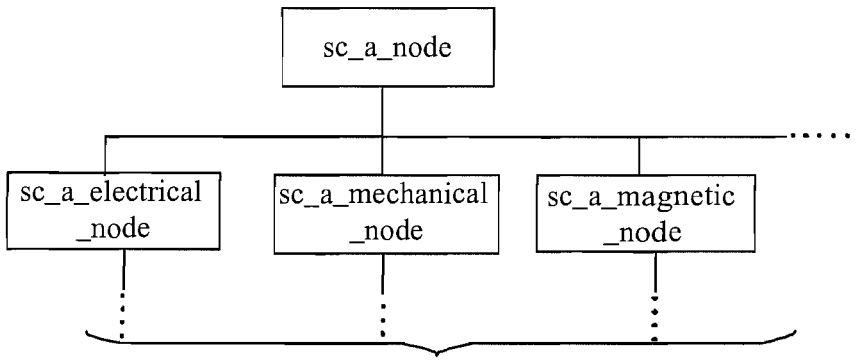

FIGURE 3.3: Analogue nodes possible inheritance hierarchy.

3.2.2 sc_a_flow

sc_a_flow is used to represent flow variables in the Modified Nodal Analysis (MNA)-like equation formulations [90] (will be explained in detail in Section 4.1.2). According to the MNA representation of some components, like a voltage source or an inductor, a current variable should be introduced in conjunction with the declaration of any of these components. Hence, the right place for a current variable to be instantiated is in the constructor of such components, e.g.:

```
inductor::inductor(...){// inductor's constructor
...
   iL = new sc_a_flow("iL");
...
}
```

3.2.3 sc_a_free_variable

The free system variable *sc_a_free_variable* is introduced to define variables when describing a system or part of it by differential and/or algebraic equations rather than a netlist of circuit components. It is useful especially when modelling systems at behavioral level for describing the functionality of system blocks. For example, Eq.3.1 is the model of a voltage controlled oscillator (VCO) which contains one free system variable named θ.

$$\dot{\theta}(t) = \frac{d\theta}{dt} = f(v) = f_c + df * V_{filter} \tag{3.1}$$

sc_a_free_variable can be instantiated in any class constructor. For the above VCO example, it could be created by the constructor of the VCO component that contains it as follows:

```
vco::vco(...){
...
   theta = new sc_a_free_variable("theta");
...
}
```

3.3 Analogue Components

Analogue circuit components (*sc_a_component*) have been developed to provide equations which describe analogue behaviour. Similarly to the system variable hierarchy, components are derived from an abstract base class. A component abstract base class contains virtual *build* method to be invoked by the analogue kernel. The *build* method will be explained in detail in Section 4.2. A sample component class hierarchy is illustrated in Figure 3.4 with examples of SPICE-like circuit elements such as resistor, capacitor, diode, and various types of autonomous sources. Arbitrary differential and algebraic equations can be included as user-defined components.

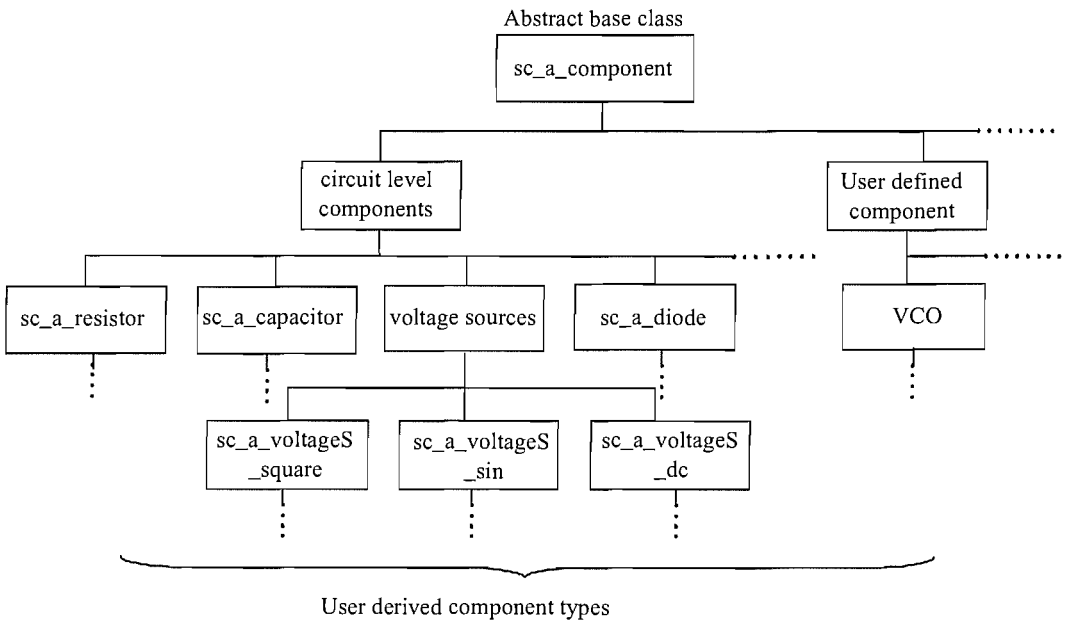

FIGURE 3.4: SystemC-A analogue components inheritance hierarchy.

A component class constructor, which defines the component's interface may contain a pair of node pointers and a value. An example of instantiating a capacitor is shown below:

```
sc_a_capacitor *c1 = new sc_a_capacitor("c1", nodeA, nodeB, C)
```

where *c1* is the component name, *nodeA* and *nodeB* are names of analogue system variable objects of type *sc_a_node* and represent the two terminals to which the capacitor is connected, and C is the capacitance. The capacitance value can be a constant or a SystemC signal, which provides a mechanism to model time-varying capacitors, as shown below:

```
sc_a_capacitor *c1 = new sc_a_capacitor("c1", nodeA, nodeB, &Cv)
```

where *Cv* is a SystemC signal of type double (*sc_signal <double> Cv*).

For user-defined components, the constructor arguments may be different and depend on the system to be modelled. Arguments can be, for example, nodes or signals, as shown below in the VCO example constructor:

```
vco *vco1 = new vco("vco1", nVfilter, &Vco);
```

where *vco1* is the name of the component, *nVfilter* is a system variable of type *sc_a_node* and *Vco* is a SystemC signal of type *bool* (*sc_signal <bool> Vco*).

An example of SystemC-A component is an inductor model as shown in Listing 3.1. The class's cpp file is defined in the code described in lines 1-32, whereas the class's header file is defined in code lines 34-49. The inductor model illustrates the format of the constructor (lines 7-11) within which necessary system variables are declared. Also, it shows the *build* method (*BuildM* and *BuildB*) which defines the component's contribution to the system matrix and will be explained in detail in Section 4.2.

```
1  #include "sc_a_inductor.h"
2  #include "sc_a_flow.h"
3
4  //inductor class constructors
5  sc_a_inductor::sc_a_inductor(){}
6
7  sc_a_inductor::sc_a_inductor(char nameC[5], sc_a_system_variable *node_a,
8  sc_a_system_variable *node_b, double value):
9     sc_a_component(nameC,node_a, node_b, value){
10    iL = new sc_a_flow("iL");//instantiate system variable of type sc_a_flow in
11 }                    //conjunction with an inductor according to MNA formulation
12
13 void sc_a_inductor::BuildM(void){// BuildM to add inductor's
14                                  // contribution to Jacobian
15    L = value;
16    Jacobian(a, iL, 1);           // inductor has 5 elements to be contributed
17    Jacobian(b, iL, -1);
18    Jacobian(iL, a, 1);
19    Jacobian(iL, b, -1);
20    Jacobian(iL, iL, -S*L);
21 }
22 void sc_a_inductor::BuildB(void){//BuildB to add inductor's
23                                  //contribution to Right hand side
24    L = value;                    //inductance
25    S = Sn();                     //get discretisation operator S
26    Lidotn=Xdot(iL);              //get the derivative
27    IE = X(iL);
28    vba = X(b)-X(a);
29    BuildRhs(a, -IE);             //add contribution to Right hand side
30    BuildRhs(b, IE);
31    BuildRhs(iL, vba + Lidotn);
32 }
33
34 // inductor header file
35 #include "sc_a_component.h"
36
37 class sc_a_inductor:public sc_a_component {
38 public:
39    sc_a_inductor();
40    sc_a_inductor(char nameC[5],sc_a_system_variable *node_a, sc_a_system_variable
41                  *node_b,double value);
42    virtual ~sc_a_inductor();
43    void BuildB(void);
44    void BuildM(void);
45
46 protected:
47    double S, Lidotn, L, IE, vba;
48    sc_a_system_variable *iL;
49 };
```

LISTING 3.1: Typical analogue component class, an inductor.

The component base class constructor attaches each newly created component to a global linked-list of system components as shown in Figure 3.5 to form a connected circuit. The list is used at the matrix build time in scanning all the components to invoke their *build* functions.

A netlist of an analogue circuit can be constructed by declaring system variables of type node and analogue components as shown in Listing 3.2 of the loop filter in

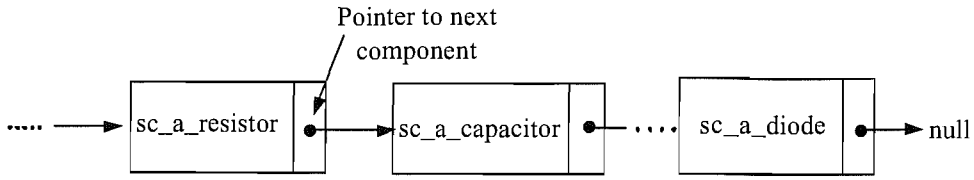

FIGURE 3.5: Linked-list of analogue components.

a phase locked loop. Figure 3.6 shows its corresponding schematic. The circuit's database is constructed once, prior to a simulation.

```
1  // instantiate nodes and components
2      n2 = new sc_a_node("n2");
3      n0 = new sc_a_node("0");
4      n1 = new sc_a_node("n1");
5      sc_a_currentS_dc *I1 = new sc_a_currentS_dc("I1", n1, n0, &Iin);
6      sc_a_capacitor *c1 = new sc_a_capacitor("c1", n1, n2, 3e-9);
7      sc_a_resistor *r1 = new sc_a_resistor("r1", n2, n0, 1e3);
8      sc_a_capacitor *c2 = new sc_a_capacitor("c2", n2, n0, 4e-9);
```

LISTING 3.2: Components and nodes instantiations forming an electronic circuit in SystemC-A.

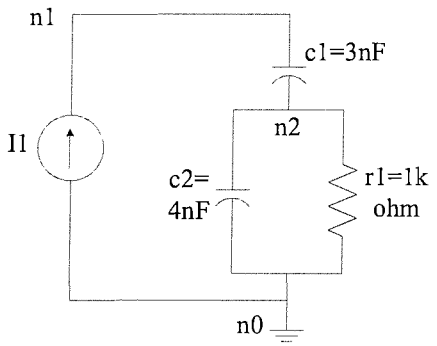

FIGURE 3.6: Corresponding schematic of circuit description in Listing 3.2.

Appendix C.1 lists other SystemC-A component's models developed within this project such as a resistor, diode, MOSFET transistor, capacitor, and different types of voltage sources.

3.4 Digital-Analogue Interactions

In modelling mixed-signal systems digital-analogue interactions are unavoidable. Connectivity between analogue and digital models requires special consideration

since the two models have different language representations. The solution to this problem is to insert a special interface model directly between the digital and analogue parts. These interface models have no corresponding physical parts. The intended interfacing solution is similar to those adopted in VHDL-AMS and Verilog-AMS. A/D and D/A interfaces are used only to change representations of signals between the digital and analogue domains. SystemC-A includes explicit interfaces modules readily available for the user. Alternatively, the high expertise user may write his interfaces hidden within the model. An example of an AMS system with explicit A/D and implicit D/A interface models is shown in Listing 3.3 and Figure 3.7 which represent a switched mode power supply (SMPS) testbench. The SMPS example is discussed in detail in Chapter 6. The following two Sections 3.4.1 and 3.4.2 will explain the functions of these interfaces.

```
1  ...
2  //SMPS mixed-signal model
3   sc_signal<bool> Vd1; //instantiate digital and analogue signals for connection
4   sc_signal<double> Va1, Va2;
5
6      sc_a_interfaceDA D_A("D_A"); //instantiate digital-analogue interface module
7      D_A.IND(Vd1);
8      D_A.OUTA(Va1);
9      D_A.clk(Clk);
10
11     digital digital1("digital1");//instantiate digital module and bond its ports
12     digital1.Vd_in(Va2);
13     digital1.VcontD(Vd1);
14     digital1.clk(Clk);
15
16     analog analog1("analog1");//instantiate digital module and bond its ports
17     analog1.Vout(Va2);
18     analog1.VcontA(Va1);
19     analog1.clk(Clk);
20 ...
```

LISTING 3.3: Explicit D/A and implicit A/D interfaces in SMPS testbench.

3.4.1 Digital-Analogue Interface

sc_a_interfaceDA shown at line 6 of Listing 3.3 is a SystemC module which contains an input port of type *bool* and an output port of type *double*. *sc_a_interfaceDA* ports are connected to signals of the corresponding types. A digital signal coming from a digital module is transformed into an analogue signal and directed towards

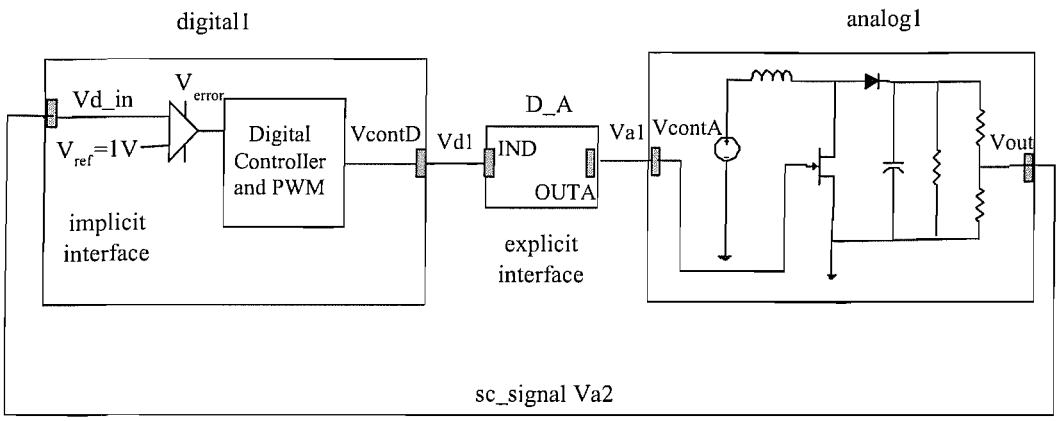

FIGURE 3.7: SMPS Block diagram with analogue-digital interfaces.

the analogue module through the output port. The simplest form of the interface is a set of switched ideal voltage sources. However, instability may be introduced in the analogue simulation due to large instability changes in node voltage when the digital node switches. Therefore, rather than changing abruptly, a transformation is done by a smoothing function explained in detail in Section 3.4.5.

3.4.2 Analogue-Digital Interface

Analogue-digital interface could be implicit as in Figure 3.7 or explicit and called *sc_a_interfaceAD*. *sc_a_interfaceAD* is a SystemC module takes an analogue signal of type *double* and produces a digital *bool* signal. The criteria to generate a digital event is simple and demonstrated in Figure 3.8. If the threshold voltage E defined is exceeded, an event with a state (high) is generated. An event with a state (low) is produced, if the analogue voltage falls below the threshold voltage. Due to the fact that the result is a digital boolean signal, an event is to be generated at every signal change. The digital part will react to this event if a concurrent statement reads this signal or if the sensitivity list of a process contains this signal.

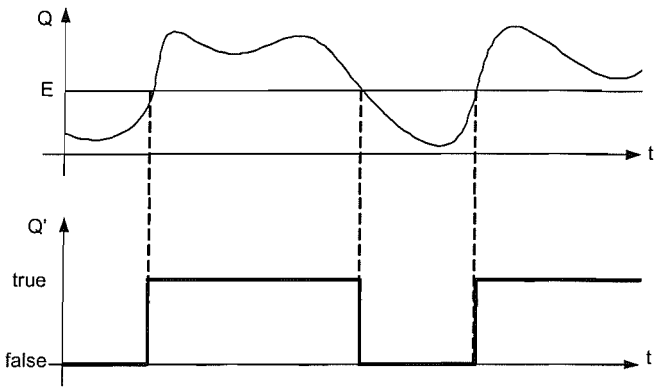

FIGURE 3.8: Demonstration of analogue to digital transformation at their interface.

3.4.3 Other Interfacing Methods

In Sections 3.4.1 and 3.4.2 A/D and D/A interfaces that connect analogue and digital signals (*sc_signal*) are illustrated. SystemC-A offers more connection implementations. It supports node to signal, signal to node, or node to node connections. Node-signal connection is used by reading a value at a node and then writing it to a SystemC signal which may be connected to another module.

```
value=X(a); // read value from node
aSig.write(value); // write value to signal
```

Node-node connection is used when there are two electronic circuits in two different modules and need to be connected via a shared node. It can be done by declaring a new node in the testbench and then do the same procedure as when connected signals in a testbench. Signal-node connection is used when a SystemC signal is used in a circuit component inputs, it can be done by reading the signal into its constructor's argument. All interfaces methods are used and tested by modelling a variety of examples in Chapter 6.

3.4.4 Analogue Stepping

The time step of the analogue simulator is usually determined by the internal algorithm of the simulator, which means it cannot be defined by the user but sometimes there is an option for the user to determine the required fixed time step. Analogue simulators do not use events but instead employ an entirely different approach to time step control, namely, continuous step size adjustment, illustrated in Figure 3.9, where $h = h_n, h_{n+1}, ...$ may have different values. This approach is used in most analogue simulators for obvious reasons. Firstly, the variable step approach can minimise the errors caused by the numerical integration methods used to solve differential equations in the circuit model. Secondly, simulation times are significantly shorter than those in a fixed step size approach. For example, for a step response, in a transient phase a small time step is needed to capture the details of the fast part of the transient while in the later part, when things are settling down, a longer time step can be used.

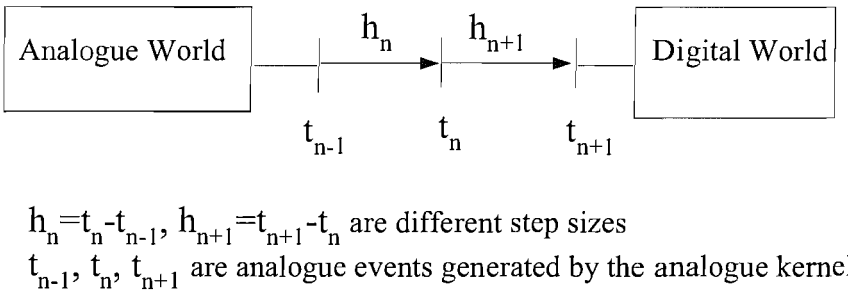

FIGURE 3.9: Time stepping in analogue simulators.

The implementation of analogue stepping is done based on the estimation of the Local Truncation Error (LTE) [91]. LTE at t_n is an error due to a numerical approximation introduced in the time point t_n. As LTE depends on the step size h_n it can be controlled by the value of h_n. The step size is determined by limiting an LTE estimate to an error bound EB_n defined as shown in Eq.3.2, 3.3,

$$|LTE|_n \leq EB_n \tag{3.2}$$

$$EB_n = RELTOL * |X_{max}| + ABSTOL \tag{3.3}$$

where X_{max} is the maximum value of a variable. $RELTOL$ is the relative error tolerance within which system variables are required to converge. It allows to simulate high and low variables without adjusting the conversion criteria. $ABSTOL$ is the absolute tolerance, it represents the smallest value of system variable that can be monitored. It forces a minimum value in the system matrix when a particular system variable is nearly zero. These are user defined parameters to determine how accurate the simulator calculates the solution. For example, RELTOL=1e-3 and ABSTOL=1e-6 for voltage, 1e-9 for current. $ABSTOL$ and $RELTOL$ can have direct impact on convergence and simulation time and have to be chosen carefully.

If the Trapezoidal method is used, which is of the second order, the upper step size bound can be determined from Eq.3.4:

$$h_n = [\frac{EB_n}{C * DD^3(t)}]^{1/3} \tag{3.4}$$

where $DD^3(t)$ denotes the 3^{rd} order divided difference approximation and $C = -\frac{1}{12}$ is the related constant for the Trapezoidal method.

In order to synchronise the analogue and digital simulators at every time point, the analogue stepping is implemented using SystemC *event* notifications. The analogue module which is responsible for calculating the above estimated value of the upper step size bound h_n notifies the digital kernel at the time point equal to (current time $+h_n$). The digital processes will be activated at this time point accordingly.

3.4.5 Small Step Sizes

When connecting analogue and digital models, critical issues and problems arise and should be handled and solved in order to simulate the whole system correctly. Examples of such issues are the handling of small step sizes and the cancellation of events.

Small step sizes may occur in D/A interfaces when a digital signal, interfaced to an analogue system variable, changes its value due to a digital event. Another case is the continuous cutting in step size by the analogue stepping algorithm in highly dynamic systems. Because of the limited precision provided by the computer's finite word length, small step sizes can cause large round-off errors and lead to inaccurate results or to non-convergence [92]. On the other hand, a zero step size may occur with repeated delta cycle which causes the system to be solved at $h = 0$ and undergo non-convergence. Small step sizes including the case of a delta cycle where $h = 0$ are handled in SystemC-A by smoothing the digital signal [93] as illustrated in Figure 3.10 using Backward Euler method. The smoothing is implemented based on Eq.3.5.

$$S'_n = \frac{S_n h_n + \tau S'_{n-1}}{\tau + h_n} \tag{3.5}$$

where S_n is the input digital signal of type *bool*. h_n is the simulation time step size. S'_n is the smoothed signal and S'_{n-1} is the past value of the smoothed signal. τ is time constant which plays as a control factor to shape the signal.

Smoothing a very short digital pulse with $\tau \geq$ than the pulse width, will transform the pulse to a short spike and the events caused by the spike will be cancelled by the simulator if the signal level is less than a logic threshold as shown in Figure 3.11. The analogue solver will prevent events from cancelling by defining a check for validity of values for both the pulse width and τ.

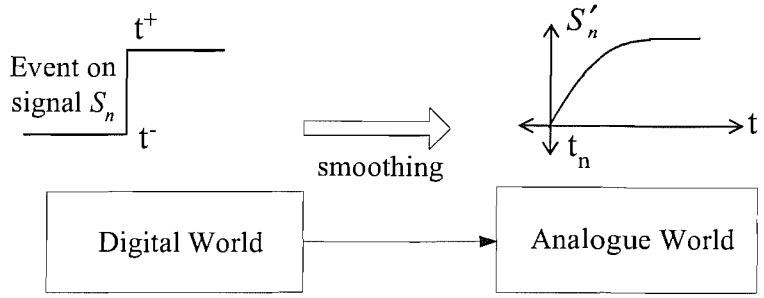

FIGURE 3.10: Handling small time step sizes in SystemC-A analogue kernel.

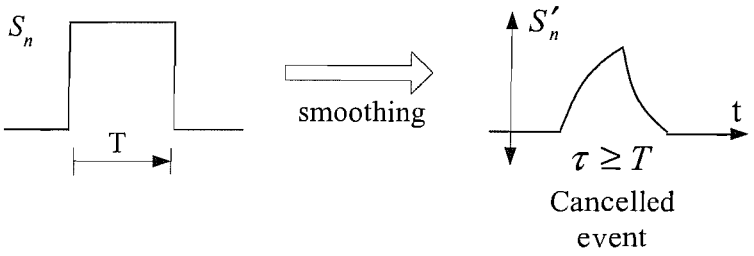

FIGURE 3.11: illustration of cancelled events when τ >pulse width.

3.5 SystemC-A Abstraction Levels

While most of the abstraction levels of the digital design flow (see Figure 1.2) are supported by SystemC, SystemC-A supports additional abstraction levels for analogue systems. SystemC-A provides methods and language constructs to support electrical circuit level and system level which includes behavioural level. Electrical circuit level is supported by the number of analogue circuit components models described in Section 3.3, which are created to be connected in SPICE-like circuit.

System level is supported by the modularity characteristic of SystemC and SystemC-A, which allow high level signal flow designs. System level also supported by the ability to co-model hardware and software. Modelling a system at behavioural level means describing the system by its DAEs. SystemC-A supports efficient nonlinear DAE formulation and solving methods.

3.6 Concluding Remarks

This chapter has presented new CAD concepts and methods for the SystemC-A AMS extensions. Novel language constructs were defined such as an abstract class for defining *system variables*. The abstract class was extended to the required types of system variable, i.e *node variable*, *flow variable* and *free variable*, to represent the unknowns in the set of DAEs. Another important abstract class is the circuit *component* which can be extended to various kinds of standard or user-defined analogue circuit components. Objects of the new elements allow building any general system such as an electronic circuit at different abstraction levels. The chapter also presents a solution to the issues of digital-analogue interaction. SystemC-A has its unique methods for interfacing analogue and digital parts represented in signal transformation and the ability to connect circuit nodes in different SystemC modules. The chapter also suggested solutions to some problems which might arise due to the interfaces such as implementing the analogue stepping, small step sizes and digital event cancellation. The proposed constructs allow modelling of analogue systems at circuit level as well as system level of abstraction, leading SystemC-A to be high level design language.

Chapter 4

Nonlinear Equation Formulation with Object-Oriented Jacobian approximation

This chapter presents the implementation of the nonlinear analogue solver which is also called *analogue kernel*. It involves two main steps to simulate systems, formulation and solving of the system's DAEs. A new method of equation formulation is proposed using C++ object-oriented features for the development of component's *build* methods. The equation formulation is called Object-Oriented Newton Quasi-Newton (OO-NQN) method. It provides means of efficient automation of equation formulation with minimal user's knowledge of the solver algorithms details.

The remainder of this chapter is organised as follows: Section 4.1 describes the numerical methods used in the implementation of the analogue kernel for formulating and solving linear and nonlinear representations of systems during a transient simulation. Section 4.2 presents the *build* methods to support equation formulation to be solved using pure Newton method, where in Section 4.3 the alternative Quasi-Newton method implementation is presented for which no Newton

derivatives are available. Section 4.6 presents the analogue kernel structure and analogue-digital modelling and simulation flow in SystemC-A. Finally, Section 4.7 gives the conclusion.

4.1 Numerical Techniques for Analogue and Mixed-Signal Simulation

The techniques of computer-aided circuit analysis are relatively well established after their rapid development in the 1960s and 1970s [94]. These techniques include equation formulation and algorithms to solve the set of system equations for different types of circuits. There are many excellent books in the literature covering the topic of circuit analysis and design, e.g. books by Calahan (1972) [95], Chua and Lin (1975) [96], Vlach and Singhal (1983) [97], Ruehli (1986) [98], as well as some recent books such as Litovski and Zwolinski (1997) [99]. The material of the following section is mainly from [99] and [100] to explain the theories behind the intended analogue kernel.

4.1.1 Mathematical Model

A nonlinear electronic circuit with lumped elements can be modelled by a set of nonlinear ordinary Differential and Algebraic Equations (DAEs) of the form:

$$\mathbf{f}(\mathbf{v}(t), \dot{\mathbf{v}}(t), t) = \mathbf{0} \quad t \geq 0, \quad \mathbf{v}(0) = \mathbf{v}_0 \tag{4.1}$$

where $\mathbf{f} : R^N \times R^N \times R^1 \rightarrow R^N$ is a vector function, $\mathbf{v}(t) \in R^N$ is a vector of unknowns, $\mathbf{v}(0)$ is a vector of initial values, $\dot{\mathbf{v}}(t)$ is a vector of unknown derivatives with respect to time, N is number of unknowns and t is time. The unknowns, which are also referred to as the primary circuit variables, are usually selected from the

set of system's node voltages, branch voltages, branch currents, capacitor charges and inductor fluxes.

Eq.4.1 can be transformed to an algebraic set by replacing the time derivatives at each time point t_n, $n = 1, 2, \ldots$, by a discrete linear differentiation formula of the general form:

$$h_n \dot{v_n} = \sum_{i=0}^{p} \alpha_i v_{n-i} + h_n \sum_{i=0}^{p} \beta_i v_{n-i} \tag{4.2}$$

where, $h_n = t_n - t_{n-1}$ is the current time step, and α_i, β_i are coefficients whose values depend on $h_n, h_{n-1}, \ldots$. The formula in Eq.4.2 is said to be of order r if,

$$LTE = h_n \, \| \dot{v_n}(t_n) - \dot{v_n} \| = O(h^{r+1}) \tag{4.3}$$

where LTE is the Local Truncation Error introduced due to the discretisation. The simplest case of the formula in Eq.4.2 is the first-order Backward Euler formula in Eq.4.4.

$$h_n \dot{v_n} = v_n - v_{n-1} \tag{4.4}$$

In this research, the Trapezoidal formula (Eq.4.5) is used. It is a popular second order form of the linear differentiation formula and the default method in SPICE and other analogue simulators.

$$h_n \frac{\dot{v}_n - \dot{v}_{n-1}}{2} = v_n - v_{n-1} \tag{4.5}$$

An example of other methods is the second-order Shichman formula [101] with variable step,

$$h_n \dot{v_n} = \frac{2h_n + h_{n-1}}{h_n + h_{n-1}} \, v_n - \frac{h_n + h_{n-1}}{h_{n-1}} \, v_{n-1} + \frac{h_n^2}{h_{n-1}(h_n + h_{n-1})} \, v_{n-2} \tag{4.6}$$

It is a particular case of the variable-order variable-step Backward Differentiation Formula (BDF) [102],

$$h_n \dot{v_n} = \sum_{i=0}^{p} \alpha_i v_{n-i} \tag{4.7}$$

BDFs are very desirable in circuit simulation due to their greater numerical efficiency and excellent stability properties in the case of stiff systems, i.e. systems with a wide spread of time constants.

It is useful to distinguish between the unknown present values of primary circuit variables v_n and the past values $v_{n-i}, i = 1, 2, ..., p$ and rewrite the formula Eq.4.2 in the following, more compact form:

$$h_n \dot{v_n} = \alpha_0 v_n + X_{v,n} \tag{4.8}$$

where the second term,

$$X_{v,n} \triangleq \sum_{i=0}^{p} (\alpha_i v_{n-i} + h_n \beta_i \dot{v}_{n-i}) \tag{4.9}$$

contains only past information and $\frac{\alpha_0}{h_n}$ is the discretisation operator (S). In this research, S and X_n are defined as in Eq.4.10 and Eq.4.11, where Trapezoidal method is used for the whole simulation time except for time=0, when Euler method is used.

$$if \quad time = 0 \quad S = \frac{1}{h_n}, \qquad else \qquad S = \frac{2}{h_n} \tag{4.10}$$

$$if \quad time = 0 \quad X_n = S v_{n-1}, \qquad else \qquad X_n = S v_{n-1} - \dot{v}_{n-1} \tag{4.11}$$

Application of the discretisation in Eq.4.8 to the original equation Eq.4.1 yields a system of algebraic nonlinear equations for every discrete time point t_n,

$$\tilde{f}(v_n) = 0 \tag{4.12}$$

Virtually all solution methods of Eq.4.12 are based on some form of the Newton-Raphson (NR) linearisation which is derived from the Taylor expansion of Eq.4.12 around the m^{th} estimation v_n^m of the solution vector v_n.

$$\tilde{f}(v_n^m + \Delta v_n^m) = \tilde{f}(v_n^m) + J^m \Delta v_n^m + ... \tag{4.13}$$

where, $J^m = \frac{\delta f}{\delta v_n}|_{v_n = v_n^m} \varepsilon R^{NxN}$ is the Jacobian matrix evaluated at v_n^m.

If the terms of order higher than the first in Eq.4.13 are rejected, then Eq.4.12 reduces to its linearised estimate,

$$J^m \Delta\ v_n^{m+1} = RHS^m \tag{4.14}$$

where the solution is $v_n^{m+1} = v_n^m + \Delta v_n^m$ and $RHS^m = -\tilde{f}(v_n^m) + J^m v_n^m$. If $\|v_n^{m+1} - v_n\| < \|v_n^m - v_n\|$ $(m = 0, 1, 2, ...)$, then the repeated application of Eq.4.14 is a contraction mapping and the subsequent values v_n^m converge to the solution v_n of Eq.4.12. The standard algorithm for the solution of Eq.4.1 (the NR algorithm) can be summarised as shown in Algorithm 4.1.

Algorithm 4.1: Newton Raphson non-linear solver. (n:time point counter, m:NR iteration counter, ϵ:arbitrary small number, T:maximum analysis time.)

```
1: n := 0;
2: t := 0;
3: repeat
4:    n := n + 1;
5:    t_n := t_{n-1} + h_n;
6:    evaluate current values of differentiation operator α_0, X_n;
7:    predict values of known variables v_n^0 as initial guess for NR iterations;
8:    m := 0;
9:    repeat
10:      m := m + 1;
11:      set up Jacobian J^m for current estimate of unknown variables v_n^m;
12:      solve linearised equation J^m Δv_n^{m+1} = RHS^m;
13:   until NR iterations converge i.e. |v_n^{m+1} − v_n^m| < ε
14:   select new step size h_{n+1} on the basis of LTE estimation.
15: until t_n > T;
```

4.1.2 Equation Formulation

The equations are ready to be formulated once they are discretised and linearised as shown in the summary in Figure 4.1 of procedural preparation of circuit simulation.

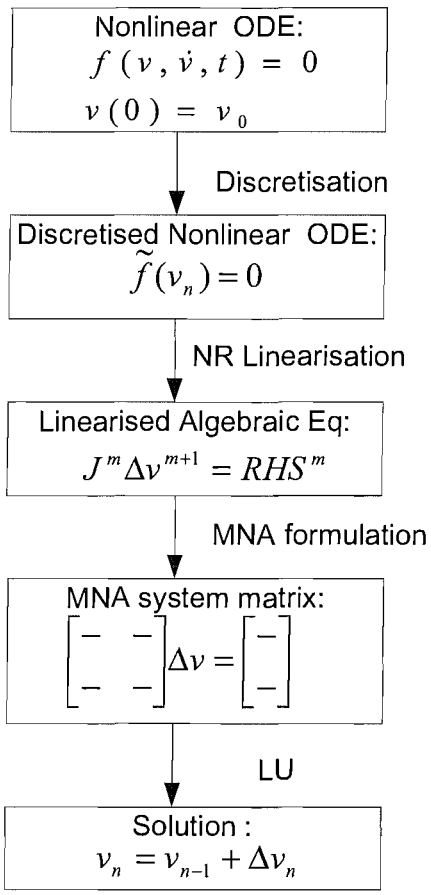

FIGURE 4.1: Procedure of analogue circuit simulation.

The analysis of the linearised system can be commonly viewed as a two stage process, equation formulation and numerical solution [99]. The Modified Nodal Analysis (MNA) method [90] has been widely used for formulating circuit equations in computer-aided network analysis. MNA retains the simplicity and other advantages of the classical nodal analysis while removing its main limitation, which is the inability to process voltage sources and current dependent circuit elements in a simple and efficient manner.

In MNA, the node equations are formulated using Kirchhoff's Current Law (KCL) in conjunction with branch constitutive equations to describe electronics circuits. The equations are formulated (represented in the computer program) automatically in a simple and comprehensive manner. The common approach for automatic equation formulation is the network element stamp method which represents the contribution of one particular element to the system of equations describing the network [99].

For example, the MNA stamp for a resistor connected between nodes i and j is,

	v_i	v_j	RHS
i	G	-G	$-i$
j	-G	G	i

(4.15)

where the rows represent equation numbers, and the columns represent variables, and G is the conductance. The Right Hand Side (RHS) represents the excitations of the linearised network. Table 4.1 shows several commonly used stamps. In the inductor stamp or voltage source stamp, a new variable is introduced representing a current through these elements. This was the main contribution of the MNA.

TABLE 4.1: Sample component stamps used in automatic equation formulation.

Resistor

	v_i	v_j	RHS
i	G	-G	$-i$
j	-G	G	i

Inductor

	v_i	v_j	I_E	RHS
i			1	$-I_E$
j			-1	I_E
I_E	1	-1	-SL	$-v_{ij} + S\dot{L}i_{En}$

Diode

	v_i	v_j	RHS
i	G_d	$-G_d$	$-I_d$
j	$-G_d$	G_d	I_d

Capacitor

	v_i	v_j	RHS
i	SC	-SC	$-C\dot{v}_n - i_c$
j	-SC	SC	$C\dot{v}_n + i_c$

4.1.3 Standard Solution of Linear Equations

Once formulated, the system of linearised equations describing a circuit has to be solved. There are two approaches to this problem: direct methods and iterative methods. Direct methods are able to solve the system in a fixed and finite number of steps, such methods are: the Gauss-Jordan method, Gaussian Elimination method and LU factorisation method. Iterative methods produce an infinite sequence of solutions that may converge to a consistent result if rather strong conditions on the Jacobian matrix are satisfied, such methods are the Jacobi method, Gauss-Seidel method and relaxation methods. In most circuit simulators the linearised Eq.4.14 is solved by means of direct methods such as Gaussian elimination or LU decomposition. These methods have proven to be reliable and accurate, thus LU is used in this research.

Nonlinear circuit analysis in the time domain may require several thousand repeated solutions of a big dimension system ($N > 500$). The linearised equations describing the circuit are usually re-formulated at each NR iteration and each time point. Therefore, the efficiency of the equation solution method and the system matrix order should be considered as an important factor. Taking advantage of system matrix properties such as sparsity and symmetry to accelerate the solution process and reduce the memory needed. Also, adopting a suitable pivoting strategy is necessary. The pivoting process consists of reordering the equations by row pivoting (renumbering the variables), column pivoting, or both, so as to put a particularly desirable element in the diagonal position from which the pivot is about to be selected. The pivoting strategy is implemented in this research but optimising system matrix by methods depending on the sparsity is not considered.

4.2 Equation Build Method

The *build* method supports the automatic OO-NQN equation formulation of the system to be modelled. It is a virtual method in the abstract component base class (*sc_a_component*) and inherited by all derived components. The *build* method consists of two functions, *BuildM()* and *BuildB()*. They contain C++ code which defines one or more DAEs. Figure 4.2 illustrates the use of the build functions in a capacitor model as a SystemC-A component. Also, it gives a better understanding of the elements of the build method and how the modeller can use them.

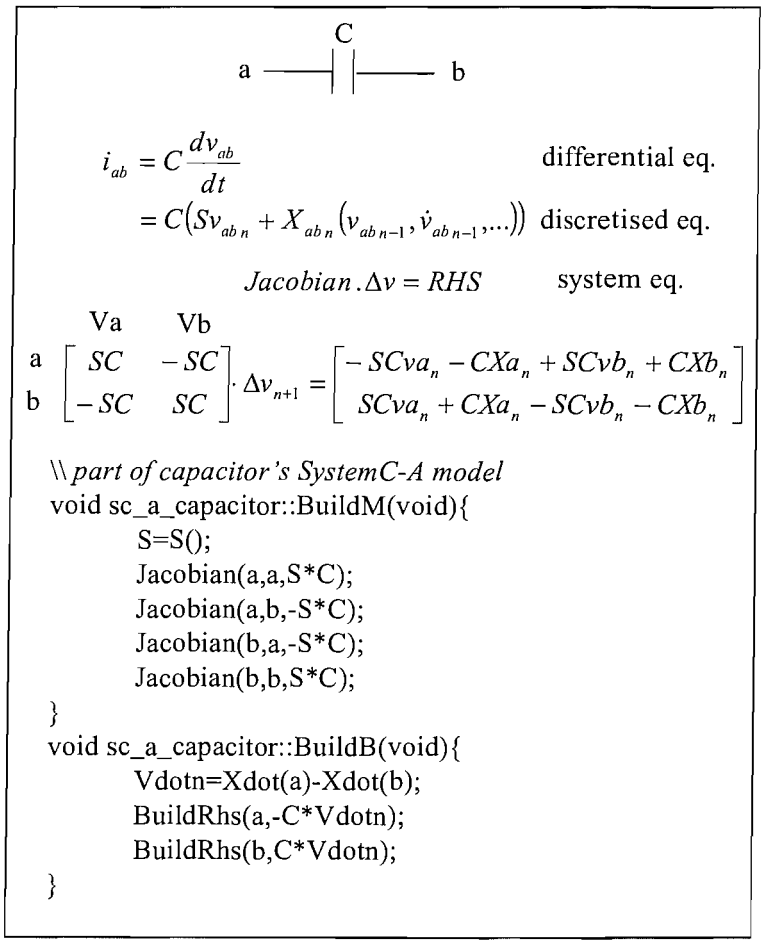

FIGURE 4.2: Capacitor mathematical model and its SystemC-A build functions.

Figure 4.2 shows first the capacitor's differential equation and its representation after discretisation (see Eq.4.10 and Eq.4.11 for the definition of the discretisation

operator S and previous terms X_n). Then it shows the capacitor's MNA stamp (Jacobian and RHS) and part of the SystemC-A model presenting the *BuildM()* and *BuildB()* functions which build the differential equation of the capacitor.

BuildM() represents the component's contribution to the Jacobian, the associated function is *Jacobian()*. It is a function to add a single contribution to the Jacobian matrix. For instance, the capacitor contribution in Table 4.1 needs four entries to the Jacobian, therefore four calls to the *Jacobian()* function is performed in the capacitor model shown in Figure 4.2. The arguments of *Jacobian()* are, first the two nodes to where the capacitor is connected and the third is the contribution element of its stamp.

BuildB() represents the component's contribution to the right hand side RHS vector of the system equation defined in Eq.4.14. The associated function with *BuildB()* is *BuildRhs()* function. It is a function to add a single contribution to the RHS of the system equation. The arguments of *BuildRhs()* are, first the corresponding capacitor node and the second is the contribution element of its stamp. If the user modelled his system using *BuildM()* and *BuildB()*, the analogue kernel will build exact Jacobian values and will solve the system using pure NR method (shall be called Newton for simplicity from now on).

Calls to *BuildM()*, which build the corresponding Jacobian entries are optional. If these calls are not provided, the solver will build the Jacobian using a secant approach with finite difference approximation of the Jacobian entries. This option will be described in Section 4.3.

The resulting Jacobian stamp conforms to the MNA formulation. The entire equation set is formulated automatically at each Newton iteration by scanning the linked-list of components (see Figure 3.5) and invoking their build methods.

4.3 Object-Oriented Jacobian Approximation

The use of pure Newton method by employing the combination of *BuildM()* and *BuildB()* to build the exact Jacobian, requires a model developer who has knowledge about the Jacobian stamps in Table 4.1. This is not always the case. Although using pure Newton method leads to accurate solutions of a simulated system, the use of *BuildM()* is optional. *BuildM()* has a default body, which calculates Newton derivative approximations using the right hand side. In this case, the modeller defines his model using only *BuildB()* and the analogue kernel will approximate the required Jacobian automatically.

The body of *BuildB()* is generated from user-defined equations. The user of SystemC-A can use *BuildB()* function only with its associated function *Equation()*, where the arguments of *Equation()* function are the same as *BuildRhs()*. The RHS vector will be provided by the modeller, while the Jacobian matrix will be estimated and then solved using Quasi-Newton method. Figure 4.3 summaries the OO-NQN equation formulation method.

The model developer may override the default *BuildM()* by providing code to calculate pure Newton derivatives. In VHDL-AMS there is no provision for user-defined derivatives so the default option must be used. However, vendor components built into a VHDL-AMS system may use pure Newton derivatives in the way described here.

4.3.1 Quasi-Newton Method

In different versions of Newton's method, if the Jacobian is not readily available and rather is approximated, the methods called Quasi-Newton methods [103]. The Jacobian matrix is composed of partial derivatives as in Eq.4.16, therefore if it is not provided each derivative element should be approximated using finite

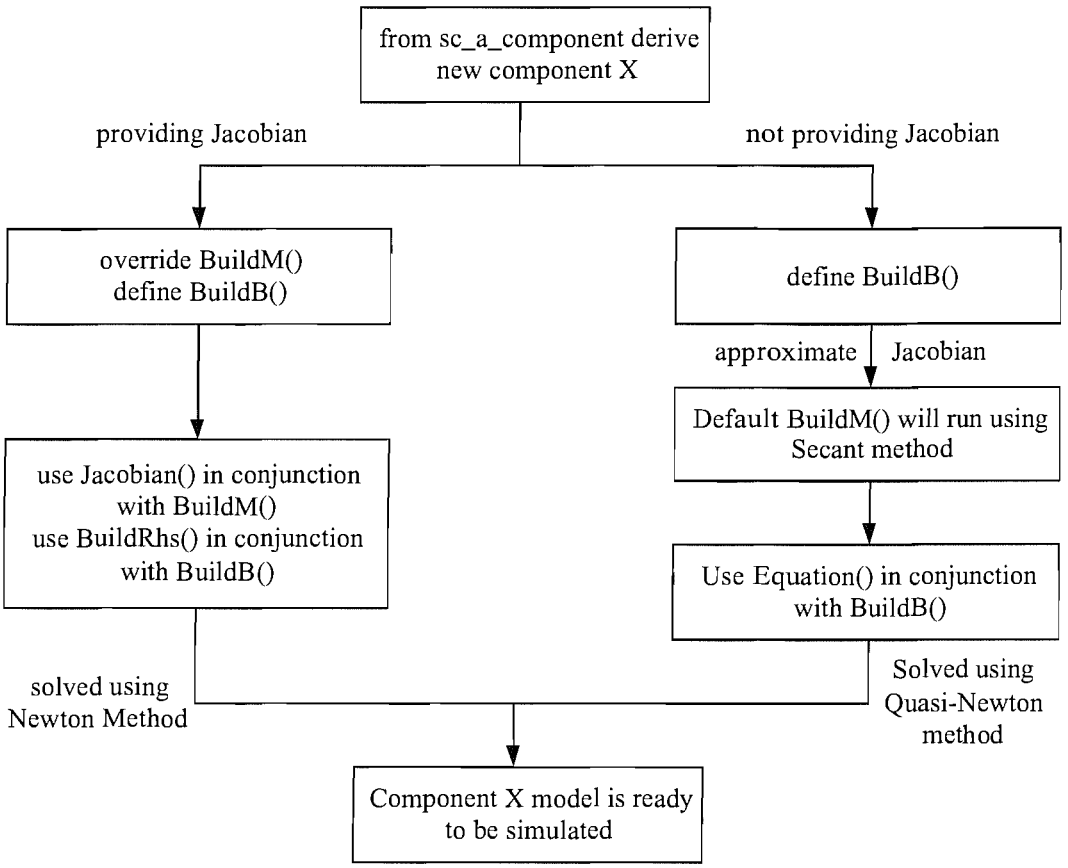

FIGURE 4.3: Summary of OO-NQN equation formulation method.

difference derivative instead of the exact one.

$$J = \begin{bmatrix} \frac{\delta f_1}{\delta x_1} & \frac{\delta f_1}{\delta x_2} & \cdots & \frac{\delta f_1}{\delta x_n} \\ \frac{\delta f_2}{\delta x_1} & \frac{\delta f_2}{\delta x_2} & \cdots & \frac{\delta f_2}{\delta x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta f_n}{\delta x_1} & \frac{\delta f_n}{\delta x_2} & \cdots & \frac{\delta f_n}{\delta x_n} \end{bmatrix} \tag{4.16}$$

Finite difference approximation is defined in Eq.4.17. It makes use of the system RHS ($f_i(x_j)$) and a scalar Δx_j which is normally chosen between 10^{-6} and 10^{-3}.

$$J_{ij} = \frac{\delta f_i}{\delta x_j} = \frac{f_i(x_j + \Delta x_j) - f_i(x_j)}{\Delta x_j} \tag{4.17}$$

This approximation method of the Jacobian is using the secant method [96]. It is

to use the slope of the line between two consecutive search points x_j and $x_j + \Delta x_j$ to compute the next point x_j^{k+1} (Eq.4.18) from secant that drawn between the two points, as demonstrated in Figure 4.4.

$$x_j^{k+1} = x_j^k + J_{ij}^{-1} f(x_j^k) \tag{4.18}$$

The advantage of the method is that the number of function's evaluations is half of that of the Newton's method because Jacobian need not be evaluated. However, its convergence rate is slightly less than Newton.

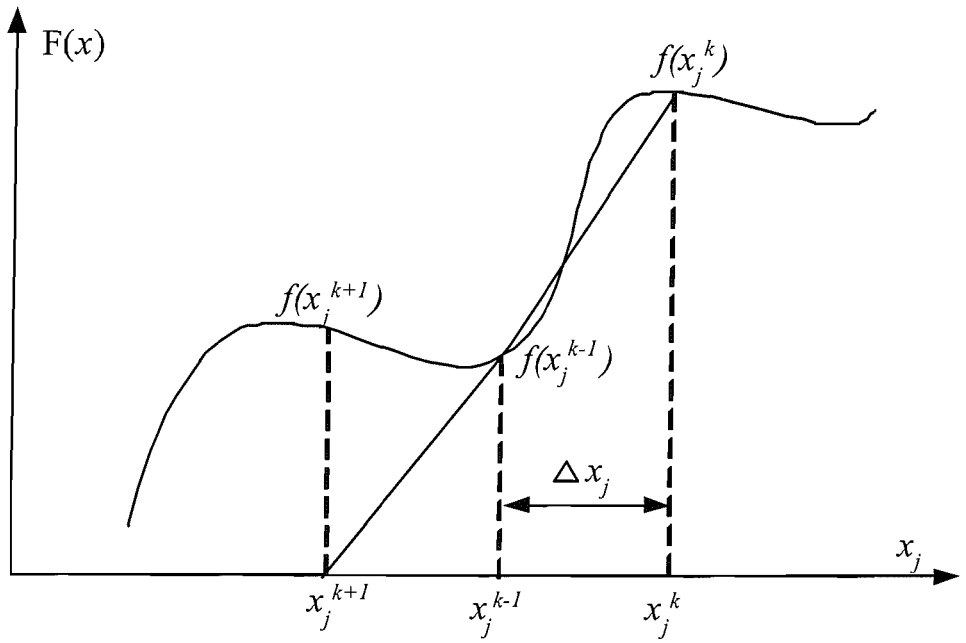

FIGURE 4.4: Illustration of the secant method.

4.4 SystemC-A Implementation of OO-NQN Equation Formulation

SystemC-A utilised the advantage of object-oriented C++ inheritance and polymorphism (as explained in Section 3.1) to implement the automation of OO-NQN equation formulation method. This was achieved by defining *BuildM()*

and *BuildB()* functions in the *sc_a_component* abstract class, as virtual methods. *BuildB()* is defined as a pure virtual method as in Line 10 of Listing 4.1. This means that *BuildB()* needs a body to be defined in any derived user's component model, otherwise the simulator will produce an error.

Most importantly, *BuildM()* virtual method defined in line 11 of Listing 4.1 has a default body. If the user does not provide the Jacobian by defining *BuildM()* in his model, the default body in the abstract component class will be run instead. The default *BuildM()* as implemented in Lines 59-86 of Listing 4.1 is the secant method approximation of the Jacobian.

```
1  //component's header file
2  #include "sc_a_system_variable.h"
3
4  class sc_a_component{
5  public:
6      sc_a_component();
7      sc_a_component(char nameC[5],sc_a_system_variable *node_a,sc_a_system_variable
8                     *node_b,double valueC);
9      virtual ~sc_a_component();
10     virtual void BuildB(void)=0;// pure virtual method (must override)
11     virtual void BuildM(void);
12     void setNext(sc_a_component *link) {next = link;};
13     sc_a_component *getNext() {return next;};
14     virtual void IC(void){}
15     void Jacobian(sc_a_system_variable *Q1, sc_a_system_variable *Q2, double value);
16     void BuildRhs(sc_a_system_variable *Q1, double value);
17     void Equation(sc_a_system_variable *Q1, double value);
18
19 protected:
20     char name[15];
21     bool flag;
22     double *x_vector, *RHS, *rhs1, *rhs2, *rhs3;
23     double Jacob, deltaX, value;
24     int i,j,n, i2, i3;
25     sc_a_system_variable *a,*b;
26     sc_a_component *next;
27 };
28
29  //component's cpp file
30  ...
31  sc_a_component::sc_a_component(char nameC[5],sc_a_system_variable
32  *node_a,sc_a_system_variable *node_b,double valueC){
33     strcpy(name,nameC);
34     a=node_a;
35     b=node_b;
36     value=valueC;
37     if (mylist != NULL)
38         mylist->insert((sc_a_component *)this);
39     else
40         throw "mylist is empty";
41     deltaX=1e-3;
42     flag=false;
43  }
44
45  void sc_a_component::Jacobian(sc_a_system_variable *Q1, sc_a_system_variable *Q2,
46  double value){
47         LS1->BuildM(Q1->get_colomnNo(), Q2->get_colomnNo(), value);
```

```
48  }
49
50  void sc_a_component::BuildRhs(sc_a_system_variable *Q1, double value){
51      LS1->BuildRhs(Q1->get_colomnNo(), value);
52  }
53
54  void sc_a_component::Equation(sc_a_system_variable *Q1, double value){
55      RHS=build1->get_RHS();
56      RHS[Q1->get_colomnNo()]=value;
57  }
58
59  void sc_a_component::BuildM(void){//approximate Jacobian using secant method
60      ...
61      for(i=1;i<n;i++){
62          LS1->BuildRhs(i,RHS[i]);//build Rhs with f(all variables updated)
63          rhs1[i]=RHS[i]; // save values of RHS of all component contributions
64          rhs2[i]=0;
65          RHS[i]=0;
66      }
67      BuildB();                   //build without variable increment
68      for(i3=1;i3<n;i3++){        //save this component contribution to rhs
69          rhs2[i3]=RHS[i3];
70          RHS[i]=0;
71      }
72
73      for(i=1;i<n;i++){           //special update, update single variable at a time
74          x_vector[i]=x_vector[i] + deltaX;
75          BuildB();               //build with variable increment
76          for(i3=1;i3<n;i3++) //save this component contribution to rhs after updating
77              rhs3[i3]=RHS[i3];
78
79          x_vector[i]=x_vector[i] - deltaX;//restore x
80          for(j=1;j<n;j++)
81          {
82              Jacob=(rhs3[j]-rhs2[j])/deltaX;
83              LS1->BuildM(j,i,-Jacob);//take values to LS
84          }
85      }
86  }
```

LISTING 4.1: SystemC-A model of the Component abstract class.

The developed object-oriented secant algorithm can be summarised in Algorithm 4.2.

Algorithm 4.2: Object-oriented secant algorithm of approximating the Jacobian.

1: evaluate equations (run BuildB()) with all variable updated to get $f(x)$ vector for a component.
2: **for** each variable x_j **do**
3: choose suitable Δx_j according to x_j magnitude
4: $x_j = x_j + \Delta x_j$ increment x_j
5: evaluate equations (run BuildB()) to get $f(x_j + \Delta x_j)$ vector
6: $x_j = x_j - \Delta x_j$ restore x_j
7: **for** each row i **do**
8: $J_{ij} = \frac{f_i(x_j+\Delta x_j)-f_i(x)}{\Delta x_j}$
9: **end for**
10: **end for**

4.5 Object-Oriented Jacobian Approximation Efficiency

All the examples modelled in Chapter 6, 7 and 8 were tested using both exact and approximated Jacobians of the OO-NQN method. The models with approximated Jacobian were as accurate as with the exact one, the percentage error between models simulated using both methods is negligible. With regard to the CPU time, Table 4.2 shows execution times of different case studies for both exact and approximated Jacobian methods.

TABLE 4.2: Execution times when using exact and approximated Jacobian in case studies covered in Chapter 6, 7 and 8.

example	time step [s]	duration [s]	samples	exact J [s]	approxi. J [s]	increase
Van Der Pol Oscillator	0.1	150	1500	0.05	0.07	1.40
Lorenz Chaos	0.01	150	15000	0.69	0.99	1.44
PLL noise 2	$0.2n$	200μ	10000	72.91	80.99	1.11
Ferromagnetic hysteresis	0.1	110	1100	0.14	0.17	1.22
Automotive, OC	$0.1m$	2.5	25000	27.35	50.55	1.84
Automotive, PIC	$0.05m$	2.5	50000	60.69	110.61	1.82
Automotive, VSC	$0.1m$	2.5	25000	63.24	112.23	1.77

The simulations were carried out on a PIII PC with 512MB RAM. It is apparent from Table 4.2 that case studies using the Jacobian approximation took longer simulation times than the ones that using exact Jacobian. Approximating the Jacobian has added a number of operations due to functions calls which increased the CPU time of modelling a particular system. Function's calls are mainly of calling the RHS at different time points in order to evaluate Eq.4.17 as described in Algorithm 4.2. Different percentage increases in Table 4.2 of different examples are due to the size of the system, i.e. number of system variables. For instance, the automotive examples are very big and contain around 13 variables, therefore the increase in CPU time is quiet big. Using exact Jacobian approach has the advantage of simulation speed over the approximate Jacobian approach, leaving the choice between simulation speed and easiness to the model developer. This

choice is not provided to modeller using popular AMS HDLs, such as VHDL-AMS and Verilog-AMS.

4.6 Analogue Kernel

The analogue kernel is the core engine of any analogue simulator. It consists of layers of several algorithms. The main functionality of the analogue kernel is to solve the set of non-linear dynamic equations. The theories and algorithms behind it are explained in detail in Section 4.1. The flow chart in Figure 4.5 shows the details of the modelling and simulation flow of a general AMS system in SystemC-A. The following paragraphs are to explain Figure 4.5.

The simulation starts by the initialisation phase where the components constructors in the user code run first, initialising all variables. Some components may have initial conditions which are scanned at very early stage by using the linked-list in Figure 3.5.

Then, after coding the system model in different modules and construct the whole system in a testbench, the simulation is started by executing the following SystemC command which must be provided in the testbench directly after the user code:

sc_start();

sc_start() may have different arguments to specify the simulation time.

The engine is now ready to start formalising and solving the defined model using universally adopted and well established methods such as LU factorisation and Newton algorithms. The Newton nonlinear solver is first initialised (iteration $z = 0$) and then started. In every Newton iteration, the analogue components list is scanned to invoke their *build* functions which add the component's contributions to the Jacobian (J) and RHS of Eq.4.14.

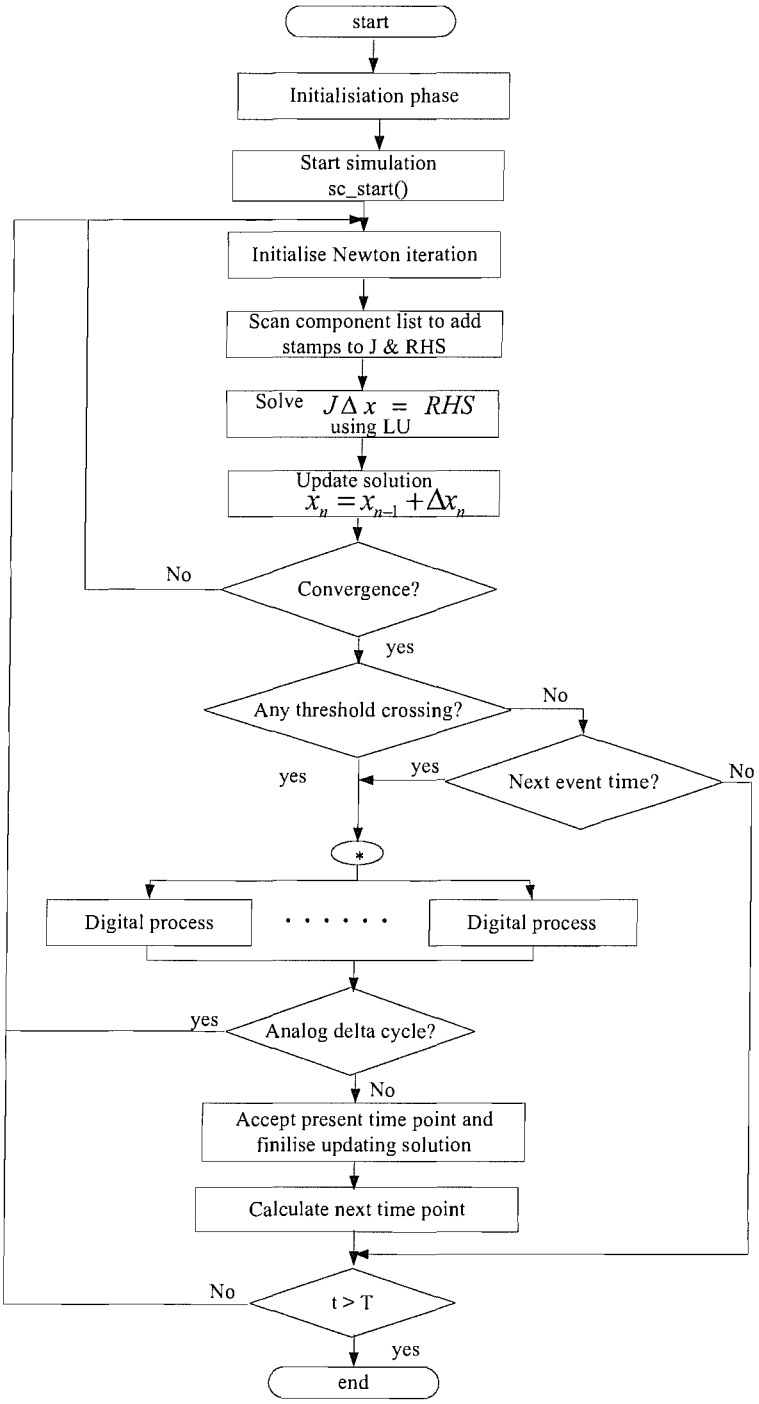

FIGURE 4.5: SystemC-A modelling and simulation.

From the iterative loop of the Newton algorithm, the LU factorisation method is invoked. LU factorisation method is responsible for solving the linear set of equations (Eq.4.14) formulated at each Newton iteration for Δx. The LU method employs a pivoting method to avoid zero diagonal element or non-convergence problems

due to small pivots. The basic pivoting strategy implemented in SystemC-A is, for each matrix's row/column find the largest element and then interchange this row/column with the current row/column. The reordered matrix should have all largest elements on its diagonal. The entire matrix must be reordered at each step of the factorisation.

Once the system equations Eq.4.14 are solved for Δx, the solution x is updated ($x = x + \Delta x$) and tested for convergence. If the convergence condition is not satisfied the Newton algorithm continues iterating. If the convergence condition is satisfied, the analogue solver exits the Newton algorithm and the digital kernel proceeds with the digital processes if required which might involve multiple delta cycles. After the present time point is accepted and the solution is updated, the next time point is calculated using the current LTE estimate. Afterwards the solver schedules an event at the next time point. The current time in SystemC is obtained from *sc_time_stamp().to_seconds()*, thus the event is scheduled at,

$$t_{n+1} = sc\_time\_stamp().to\_seconds() + h_n \tag{4.19}$$

where h_n is the next time step. This process is discussed in more detail later in Section 5.4. Any of the triggered digital processes might trigger the analogue kernel and the cycle starts again.

4.7 Concluding Remarks

This chapter has first presented the numerical methods required to construct the analogue kernel. In this context, it reviewed the DAE's mathematical model, discreteisation, linearisation, formulation and solving. Then, the chapter presented the Object Oriented Newton-Quasi Newton method (OO-NQN) for equation formulation. OO-NQN was implemented by introducing a new SystemC-A *build*

method. OO-NQN has two approaches for constructing the Jacobian, building the exact Jacobian's elements using MNA component's stamps or approximating them using secant method. The advantages of OO-NQN is a compromise between simpler model or simulation speed. Finally, the analogue kernel with the developed constructs from Chapter 3 are put together to form an engine to simulate AMS systems.

Chapter 5

Time Synchronisation Between Analogue and Digital Kernels

One of the most important problems in mixed-signal simulation is the time synchronisation between the event-driven digital simulation and numerical integration in the analogue solver. Synchronisation is an essential issue affecting the simulation speed and accuracy. Another important issue to consider is the signal conversion on mixed-nets at the analogue-to-digital and digital-to-analogue interfaces. This is explained in detail in Chapter 3.

The idea of synchronisation is to modify the time stepping engine in the analogue solver such that it fits into the digital event-driven paradigm. The synchronisation is then accomplished by the digital simulator, which processes the events in the chronological order of their time stamps. For this purpose, a new implementation of the lock-step method, with efficient handling of zero step-sizes is used.

The remainder of this chapter is organised as follows. Section 5.1 begins with a description of the SystemC digital simulation cycle. Section 5.2 describes the SystemC-A mixed-signal simulation cycle, in which the AMS extensions are handled as a modification to the original SystemC kernel. In Section 5.3, various

synchronisation methods from the literature with their advantages and disadvantages are discussed. In particular, the lock-step approach is chosen to be further investigated and implemented in this research, as described in Section 5.4. Finally, Section 5.5 concludes the chapter.

5.1 SystemC Simulation Cycle

Like in the case of most high level HDLs, a SystemC model consists of a hierarchical network of parallel processes. These processes exchange messages under the control of the simulation kernel process [10] and concurrently update the values of signals and variables. Signal assignment statements do not affect the target signals immediately, but the new values become effective in the next simulation cycle [104]. The kernel process resumes when all the user defined processes become suspended either by executing a *wait()* statement or upon reaching the last process statement. On resumption, the kernel updates the signals and variables and suspends again while the user processes resume. If the time of the next earliest event (t_n) is equal to the current simulation time (t_c), the user processes execute a delta cycle, illustration of the SystemC simulation cycle is shown in Figure 5.1.

5.2 Developed SystemC-A Mixed-Signal Simulation cycle

In SystemC-A mixed-signal simulator, the digital and analogue simulation cycles are combined into a single cycle. Hence, a set of computations of the analogue equations is executed between the digital evaluation points. To comply with the SystemC execution semantics, the proposed SystemC-A simulator comprises an

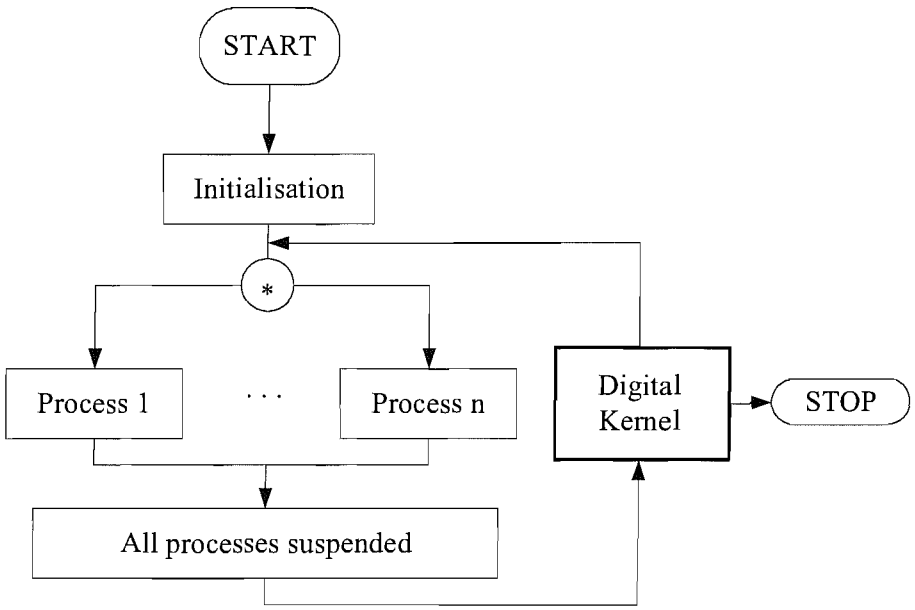

FIGURE 5.1: Simulation cycle of a SystemC model (see Section 5.1).

analogue kernel (see Figure 5.2), which is activated by the modified SystemC kernel and drives the user defined analogue descriptions.

The SystemC-A simulation cycle shown in Figure 5.2 starts with the initialisation phase, where the initial signals and analogue system variables values are computed. The initial analogue equation system is determined by running the component's function (*InitialC()*) at the initialisation phase. The simulation cycle itself starts with a computation of analogue solution points. This continues until the next digital event is scheduled or event occurs at analogue-digital interface. To compute a digital evaluation point, signals are updated. After that, processes are executed. If the time for the next digital evaluation (t_n) is equal to the current simulation time (t_c), the digital simulator is called again with the same current simulation time (delta cycle). If t_n is not equal to t_c, the analogue solver is called, and the next cycle begins. This continues until the end of the simulation is reached ($t_n = t_{end}$).

The analogue kernel repeatedly executes the simulation cycle shown in Figure 5.3, which might involve delta cycles and backtracking. The analogue solver must have

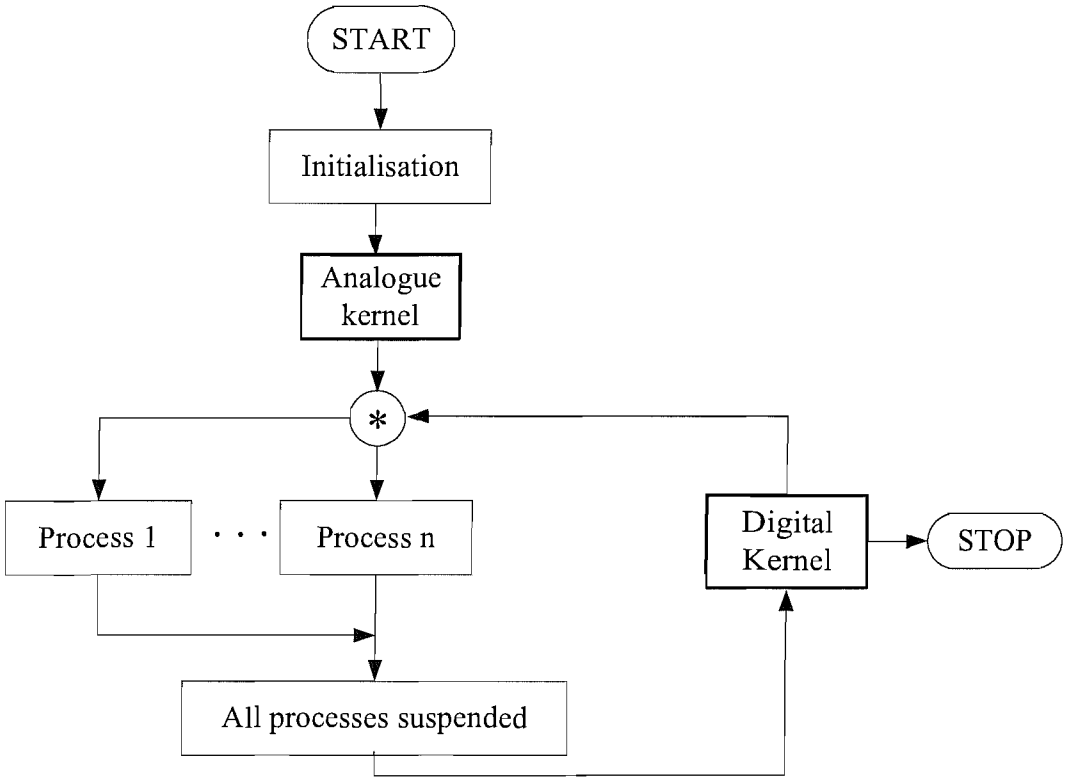

FIGURE 5.2: Proposed SystemC-A simulation cycle.

a capability to backtrack to the state t_{c-1} (just before t_c). Backtracking can be achieved by saving the analogue state at t_{c-1}.

Analogue simulators use continuous step size adjustment to minimise the errors caused by the numerical integration formula (see Section 3.4.4). It is therefore necessary for the analogue kernel in a SystemC environment to handle delta cycles in a manner similar to that of digital processes. However, the state of the analogue solver may not be updated until after the SystemC kernel advances the simulation time ahead of the current simulation time t_c, unless a delta cycle occurs and reevaluation of the current step is necessary.

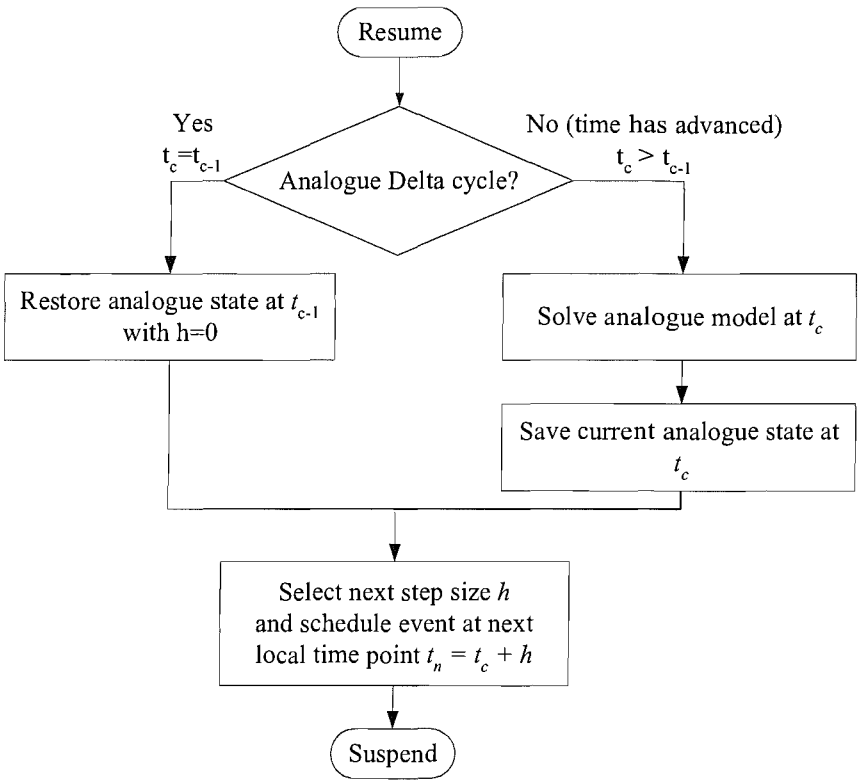

FIGURE 5.3: Simulation cycle of the analogue kernel process.

5.3 Time Synchronisation Methods

The major function of a mixed-signal simulator is synchronising the two distinct algorithms so information can be exchanged without incurring errors or undue overhead. There are two fundamental approaches to time synchronisation at the analogue and digital interfaces, pessimistic and optimistic [105, 106, 107]. In the pessimistic approach, the simulators progress with the same time step. This approach ensures that there is no need for backtracking and no results are thrown away. One well-known example is the lock-step method which is the technique used in this project.

The optimistic approach allows each simulator to progress in time until it runs out of internal events. If an event from one simulator is generated before the end of this optimistic time interval, all results generated after that event are discarded. This means that simulators must be able to backtrack. Examples of optimistic

synchronisation methods are Backplane [108], Ping-pong [105] and Calaveras [109]. The idea and the disadvantages behind each technique are shown in Figure 5.4 and explained in the following subsections.

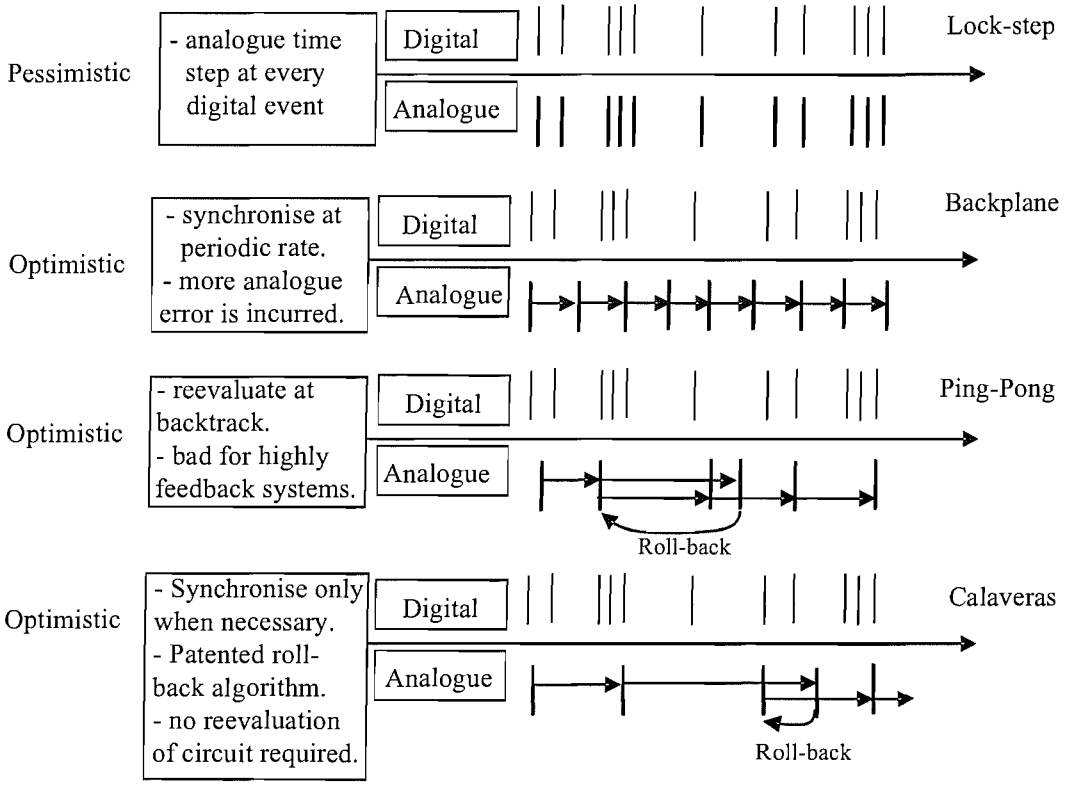

FIGURE 5.4: Time synchronisation methods of analogue and digital kernels.

5.3.1 Backplane Method

The backplane synchronisation used to be a popular method for operating multiple digital simulators concurrently using a fixed time step even if no data transfer is necessary. Initially, this approach aimed to provide the ability to run different types of digital simulator, such as VHDL and Verilog. Some simulation backplanes have attempted to attach analogue simulators [108], but performance has usually been less than optimum, because simulation backplanes typically force additional synchronisation events. Though these events may add little overhead to digital

simulators, they can cause additional analogue time steps that significantly increase overall simulation time up to 1000 to 1 ratio of analogue to digital [109]. The effect can be enormous. Usually it is common to link an analogue simulator to a digital simulator that communicates with the backplane, instead of having the analogue simulator directly connected to the backplane.

5.3.2 Ping-Pong Method

A far more efficient way to perform mixed-signal simulation is to synchronise the analogue and digital portions of the simulation only when data need to be exchanged. Otherwise, the analogue and digital algorithms work independently, each taking the optimum time steps whenever possible.

This technique is commonly referred as the ping-pong or roll-back method [105], because each portion of the simulator alternates taking time steps. The problem with this method is that one of the algorithms can get ahead of the other, which mean data needs to be exchanged at a previous time point. To maintain accuracy, the algorithm that is ahead of time must backtrack to regain synchronisation. This backtracking can be expensive in terms of CPU time because the analogue matrix must be reevaluated. With circuits that have tight feedback loops between analogue and digital, backtracking can slow down the simulation or even give erroneous results.

5.3.3 Calaveras's Method

Calaveras's Method is an improved version of the ping-pong method. The method is used exclusively by SABER simulator from Analogy [109]. When data needs to be exchanged, if the analogue simulator is ahead of time, it again rolls back in

time, but the analogue matrix instead of being re-evaluated, is interpolated at the synchronisation point.

5.4 Lock-Step Method

The alternative approach, adopted in this project, is the lock-step method. The analogue simulator calculates the step sizes and the digital simulator uses these values. The analogue kernel advances until the current simulation time and before suspending, schedules an event at the time equal to the current simulation time plus the next selected step size. Lock-step has been used by many commercial mixed-signal simulators, such as Lsim Power Analyst from Mentor Graphics and Pspice from Microsim Corporation.

The lock-step pessimistic approach has been used in preference to optimistic approaches, because the prospect of wasting vast amounts of CPU time by the optimistic approach was considered too costly [105]. Another reason is that the adopted method eliminates the need for backtracking and no results are thrown away. There were claims that the lock-step method produces long run-times [110]. However, this is true when the method is used to synchronise analogue and digital simulator from two or more different environments, because of the communication overhead. When two solvers are synchronised within the same environment, the lock-step approach is not expected to produce significant overheads. For this reason SystemC-A uses lock-step method which has been proved to be very accurate, fast, and reliable by the simulation of the examples presented in Chapter 6.

The method is implemented in this research by modifying the SystemC kernel specified by (*sc_simcontext.cpp*) module from the SystemC library. The modification is done by inserting a call to the analogue kernel before the evaluation phase of the digital simulation cycle, as shown in Listing 5.1.

```
1  void sc_simcontext::crunch() {
2  #ifdef DEBUG_SYSTEMC
3      int num_deltas = 0;
4      // number of delta cycles
5  #endif
6      m_delta_count ++;
7      while( true ) {
8  TS->TranS(); // <-- added code
9  // a call to the analogue solver
10     // EVALUATE PHASE
11 ...
12     }
13 ...
14 }
```

LISTING 5.1: Modification to the SystemC kernel to be coupled with the analogue kernel.

This approach ensures that the SystemC kernel will make a step in time no larger than the analogue kernel's step size. Since the analogue kernel is controlled by the SystemC kernel, no synchronisation deadlock may happen. The only causes for deadlock-like behaviour could arise due to a failure to converge in the analogue solver or due to unresolvable delta cycles.

Most existing digital solvers cannot backtrack and therefore no fundamental changes are required if a mixed-signal system is integrated to the SystemC kernel. The lock-step synchronisation algorithm has been implemented as a modification to the digital kernel and can be described in the form of pseudo-code as in Algorithm 5.1.

Algorithm 5.1: Lock-step synchronisation method.

```
1: time = 0
2: initialise both the analogue and digital kernels.
3: while (time <= end time) do
4:    while (immediate notifications are pending) do
5:       execute the analogue kernel
6:       distribute notifications generated by the analogue kernel on global nets.
7:       while (there are active processes) do
8:          run a selected process
9:       end while(there are active processes)
10:      update signals.
11:      check if a delta cycle is necessary
12:   end while(immediate notifications are pending)
13:   advance time to the next timed notification.
14: end while(time <= end time)
```

5.5 Concluding Remarks

The aim of this chapter was to develop a method to synchronise the developed analogue kernel in Chapter 4 and the digital SystemC kernel. In this context, the chapter explains first the SystemC digital simulation cycle and the SystemC-A simulation cycle. Further, the chapter reviewed some of the well known synchronisation methods from the literature, suggesting the lock-step method to be used in this research. The implementation of the lock-step synchronisation method relies essentially on a simple modification of the SystemC kernel. The lock-step method has proven to be fast, accurate, and reliable by simulating the examples presented in Chapter 6, 7 and 8.

Chapter 6

Electrical System Modelling Case Studies

The new methods and constructs of SystemC-A developed in Chapter 3, 4 and 5 have been validated using a wide range of examples. The examples could be analogue, mixed-signal, of different abstraction levels and from different domains. For this purpose, this chapter presents modelling four electrical case studies, ranging from simple to complex. The cases were chosen to test the simulator from different aspects. In Sections 6.1 and 6.2 the Van Der Pol oscillator and Lorenz chaos are modelled as systems of simple ODEs that demonstrate the modelling capabilities of SystemC-A at behavioural level.

In Sections 6.3 and 6.4 a Switched Mode Power Supply (SMPS) and a 2GHz Phase Locked Loop PLL-based frequency multiplier are modelled as non-trivial systems. Systems of this kind usually put standard SPICE-like simulators into difficulties because of the disparate time scales of their transients. In the case of the SMPS, the analogue transient in the output circuit is four to five orders of magnitude slower than that of the fast switching waveform in the digital controller. A typical simulation in a system of this kind might require a few million time

points. Excessive CPU times often occur when the entire system is modelled on the circuit level. The capacity of SystemC-A to enable mixed-signal modelling can vastly reduce simulation times where concepts need to be verified quickly and detailed circuit level modelling is not required.

6.1 Van Der Pol Oscillator

The Van Der Pol equation [111] is a model of a real electronic circuit studied in the 1920s, i.e. the days of vacuum tubes, by Balthazar Van Der Pol. In certain conditions, the tube acts like a normal resistor when the current is high. It becomes a negative resistor when the current is low. This behaviour leads to a relaxation oscillation. This system can also be represented as an RLC loop, but with the passive resistor replaced by an active element. The interplay between energy injection and energy absorption results in a periodic oscillation in voltages and currents.

The Ordinary Differential Equation (ODE) (Eq.6.1) that describes this system is one of the most intensely studied equations in nonlinear dynamics. It serves as a basic model of self-sustained oscillations arising in systems of mechanical and electronics engineering, biology, biochemistry, and many other areas of applications.

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0 \tag{6.1}$$

μ is a constant that affects how non-linear the system is. It can change the nature of oscillations from sinusoidal to relaxation. For μ equal to zero, the system is actually just a linear oscillator. As μ grows, the non-linearity of the system cannot be ignored. The Van Der Pol equation is of a second order. It should be transformed to a set of two first order equations in order to be solved. Let $x = y_1$

and $\frac{dx}{dt} = y_2$, this produces Eq.6.2 and Eq.6.3.

$$\frac{dy_1}{dt} = y_2 \tag{6.2}$$

$$\frac{dy_2}{dt} = \mu(1 - y_1^2)y_2 - y_1 \tag{6.3}$$

The two equations, after linearisation, contribute to the system Jacobian and RHS as follows:

$$\begin{array}{c|c} \multicolumn{2}{c}{\mathbf{J}} \\ \hline \mathrm{S} & -1 \\ \hline 2y_{1n}y_{2n} + 1 & S - \mu(1 - y_{1n}^2) \end{array} \cdot \begin{array}{c} \mathbf{\Delta y} \\ \hline y_1 \\ \hline y_2 \end{array} = \begin{array}{c} \mathbf{RHS} \\ \hline -\dot{y_{1n}} + y_{2n} \\ \hline -\dot{y_{2n}} + \mu(1 - y_{1n}^2)y_{2n} - y_{1n} \end{array} \tag{6.4}$$

6.1.1 Modelling and Simulation

The SystemC-A model of the Van Der Pol equation, shown in Listing 6.1, demonstrates the use of the new equation constructs and initial conditions function of SystemC-A. Modelling a typical equation such as Van Der Pol requires a few simple steps. Firstly, the model developer has to identify the variables of the equation set (in this case free variables which are objects of the class *sc_a_free_variable*) and define them in the class's constructor (lines 22-26). Secondly, the initial conditions have to be defined for the simulator (lines 28-31). Finally, the equation stamp is developed in the build functions (*BuildM()* and *BuildB()*) to pass the equation contributions to the Jacobian and RHS every time the model is needed.

```
1  // header file
2   ...
3   #include "systemcA.h"
4
5  //Van Der Pol class is derived from component class
6   class VanDerPol: public sc_a_component{
7   public:
8       VanDerPol();
9       VanDerPol(char nameC[5]);
10      virtual ~VanDerPol();
11      void BuildM(void);
12      void BuildB(void);
13      void IC(void);
14
15  protected:
16      sc_a_free_variable *y1, *y2;
17      double mu, S, Y1n, Y2n Y1dotn, Y2dotn;
18  };
19
20 // cpp file
21 ...
22 VanDerPol::VanDerPol(char nameC[5]):// class constructor
23      component(nameC,0, 0, 0){
24          y1 = new sc_a_free_variable("y1");//instantiate Van Der Pol variables
25          y2 = new sc_a_free_variable("y2");
26 }
27
28 void VanDerPol::IC(void){//Initial Conditions
29      InitialC(y1,0);  // x=0
30      InitialC(y2,0.1);// xdot=0.1
31 }
32
33 void VanDerPol::BuildM(void){
34      mu=1.0;
35      S=Sn();     //get discretisation operator
36      Y1n=X(y1);
37      Y2n=X(y2);
38
39      Jacobian(y1,y1,S);//add Van der Pol contribution to Jacobian
40      Jacobian(y1,y2,-1);
41      Jacobian(y2,y1,2*Y1n*Y2n +1);
42      Jacobian(y2,y2,S - mu*(1-Y1n*Y1n));
43 }
44
45 void VanDerPol::BuildB(void){
46      mu=1.0;
47      Y1n=X(y1);
48      Y2n=X(y2);
49      Y1dotn=Xdot(y1);// get derivatives
50      Y2dotn=Xdot(y2);
51
52      BuildRhs(y1,-Y1dotn + Y2n);//add Van der Pol contribution to RHS
53      BuildRhs(y2,-Y2dotn + mu*(1 - Y1n*Y1n)*Y2n-Y1n);
54 }
```

LISTING 6.1: SystemC-A model of Van Der Pol equations using exact Jacobian.

The use of *BuildM()* is optional, the user has the choice of simulating his model with same accuracy and a shorter simulation time using the exact Jacobian as shown in the model in Listing 6.1 or using the default method to approximate the Jacobian as shown in Listing 6.2. *BuildB()* is only used with conjunction of

Equation() function (more details are described in Section 4.3).

```
1  ...
2  void VanDerPol::BuildB(void){
3      mu=1.0;
4      Y1n=X(y1);
5      Y2n=X(y2);
6      Y1dotn=Xdot(y1);//get derivatives
7      Y2dotn=Xdot(y2);
8
9      Equation(y1,-Y1dotn + Y2n);//add Van der Pol contribution to RHS
10     Equation(y2,-Y2dotn + mu*(1 - Y1n*Y1n)*Y2n -Y1n);
11 }
```

LISTING 6.2: SystemC-A model of Van Der Pol equations using estimated Jacobian formed by Quasi Newton method.

The Van Der Pol equation set Eq.6.1 is simulated at behavioural level with the initial conditions $y_1(0) = 0$, $y_2(0) = 0.1$. The system was simulated with the maximum simulation time of 50 seconds to cover the transient time and several steady-state cycles. Simulation results of Figure 6.1 and Figure 6.2 have indicated close behaviour to the original Van Der Pol oscillator. Figure 6.1 shows time signals of both system variables y_1 and y_2 for $\mu = 1$. Figure 6.2 is the oscillator's phase plane, i.e. the plot of y_2 versus y_1.

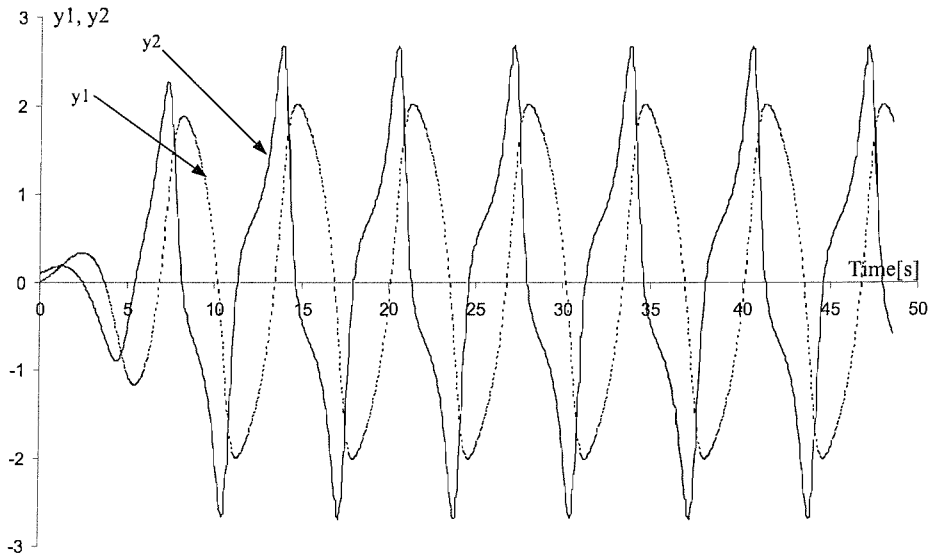

FIGURE 6.1: SystemC-A simulated time signals of Van Der Pol equation.

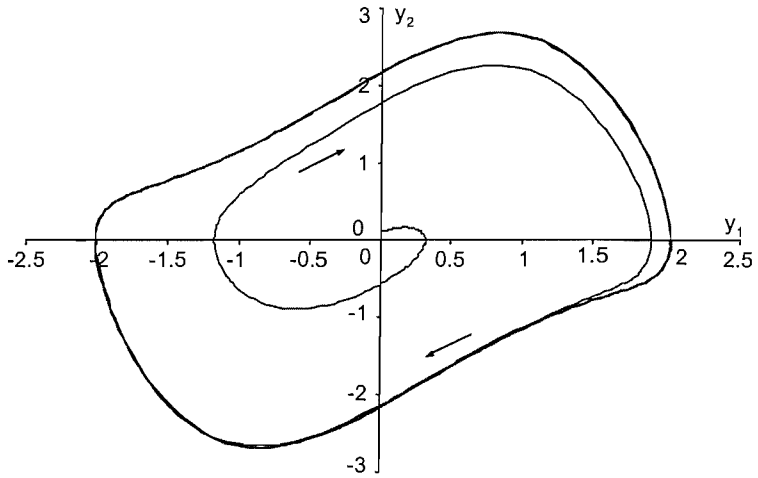

FIGURE 6.2: SystemC-A simulation of Van Der Pol equation phase plane.

The Van Der Pol equation is well known, therefore, the results can be verified by comparing them to results from other simulators such as MATLAB [112]. In addition, in C++ based languages, the user can easily export his results in text files and view [1] them or undertake more analysis in other softwares. This is a great advantage over some VHDL-AMS simulators [113] which export results in JPEG images only.

6.2 Lorenz Chaos

The so called "Lorenz attractor" [114] was first studied by Ed N. Lorenz, a meteorologist, around 1963. It was derived from a simplified model of convection flows in the Earth's atmosphere. It also arises naturally in models of lasers and dynamos. The system is expressed as the following three coupled non-linear ODEs

[1]SystemC does not include an analogue signal viewer, therefore, all graphs presented in this thesis are produced using Excel spreadsheets generated from data stored in text files using the C++ *cout* command.

(Eq.6.5, Eq.6.6, Eq.6.7):

$$\frac{dx}{dt} = \sigma(y - x) \tag{6.5}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{6.6}$$

$$\frac{dz}{dt} = xy - \beta z \tag{6.7}$$

The commonly used set of constant values is: $\sigma = 10$, $\rho = 28$, $\beta = 8/3$. Another common set is: $\sigma = 28$, $\rho = 46.92$, $\beta = 4$. σ is sometimes known as the Prandtl number and ρ is the Rayleigh number. The simulated output never reaches a steady state. Instead, it is an example of deterministic chaos. The Lorenz system is sensitive to the three constants σ, ρ, β, and to the initial conditions, a small change in the initial conditions might produce a qualitative change in the output.

Linearisation of the equations produces the following stamp for the system Jacobian and RHS.

$$\begin{array}{c|ccc} & \mathbf{J} & & \\ \hline S+\sigma & -\sigma & 0 \\ -\rho+z_n & S+1 & x_n \\ -y_n & -x_n & S+\rho \end{array} \cdot \begin{array}{c|} \mathbf{\Delta y} \\ \hline \Delta x \\ \Delta y \\ \Delta z \end{array} = \begin{array}{c|} \mathbf{RHS} \\ \hline -\dot{x_n} + \sigma y_n - \sigma x_n \\ -\dot{y_n} + \rho x_n - y_n - x_n z_n \\ \dot{z_n} + x_n y_n - \beta z_n + x \end{array} \tag{6.8}$$

6.2.1 Modelling and Simulation

The SystemC-A model of the Lorenz Chaos was developed by following the same steps as those for the Van Der Pol system, as presented in Listing 6.3. The model uses the default method of Quasi-Newton equation formulation. The class's interface (lines 2-7) has zeros in its argument indicating that the system has no inputs. The system's parameters could be passed to the model from its interface, rather they defined inside the model (lines 16-18). The initial values of the system

variables were $(x(0) = 0, y(0) = 5, z(0) = 25)$. The system was simulated with the maximum simulation time of 10 seconds. The simulated waveforms are shown in Figure 6.3 while Figure 6.4 shows the famous xz butterfly trajectory. The results were verified by a comparison to Matlab [115] simulations illustrating highly comparable figures.

```
1  ...
2  LorenzChaos::LorenzChaos(char nameC[5]):
3      sc_a_component(nameC,0, 0, 0){//instantiate Lorenz system variables
4          z = new sc_a_free_variable("z");
5          y = new sc_a_free_variable("y");
6          x = new sc_a_free_variable("x");
7  }
8
9  void LorenzChaos::IC(void){
10     InitialC(x,0.0);//initial conditions of system variables
11     InitialC(y,5.0);
12     InitialC(z,25.0);
13 }
14
15 void LorenzChaos::BuildB(void){
16     sigma=10.0;//constant values of Lorenz equations
17     rho=28.0;
18     beta=8.0/3.0;
19     Xn=X(x);
20     Yn=X(y);
21     Zn=X(z);
22     Xdotn= Xdot(x);//get derivatives
23     Ydotn= Xdot(y);
24     Zdotn= Xdot(z);
25
26     Equation(x,-Xdotn+sigma*Yn-sigma*Xn);//Lorenz equation 1
27     Equation(y,-Ydotn+rho*Xn-Yn-Xn*Zn);//Lorenz equation 2
28     Equation(z,-Zdotn+Xn*Yn-beta*Zn);//Lorenz equation 3
29 }
```

LISTING 6.3: SystemC-A Lorenz chaos model.

6.3 Switched-Mode Power Supply

Switched Mode Power Supplies (SMPS) [116] are the current state of the art in high efficiency power supplies. In this example, a boost (step-up) SMPS is used as a typical 3.3V regulator. The circuit schematic is presented in Figure 6.5. SMPS modelling is not an easy task for model developers using existing simulators. It also provides high challenges to SystemC-A, since it is a complex mixed-signal system which needs excessive CPU times when simulated. Further, it utilise many

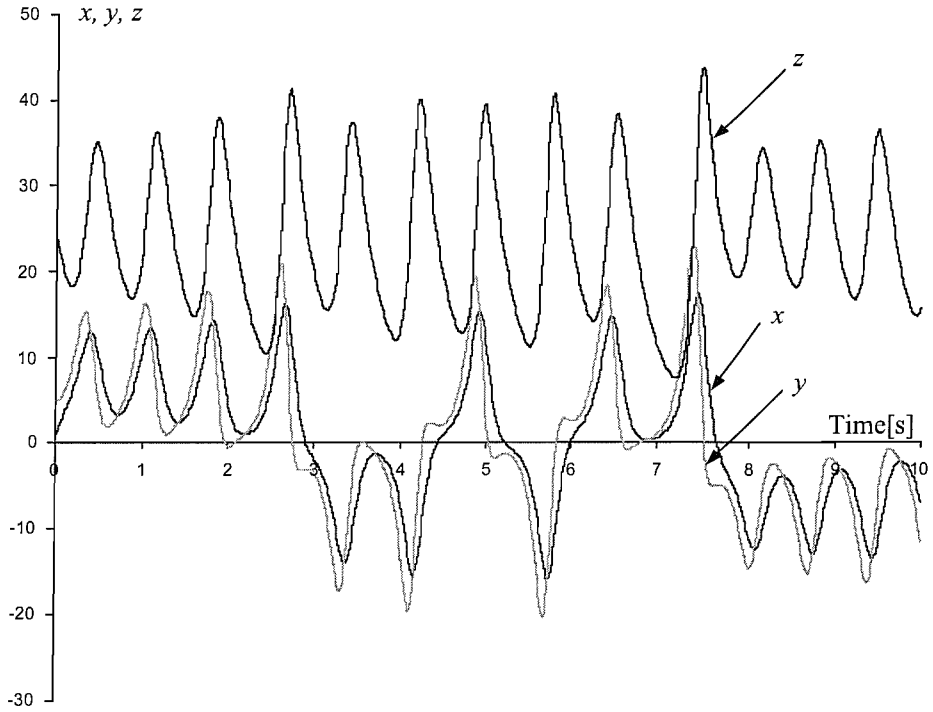

FIGURE 6.3: SystemC-A simulation of Lorenz chaos time signals.

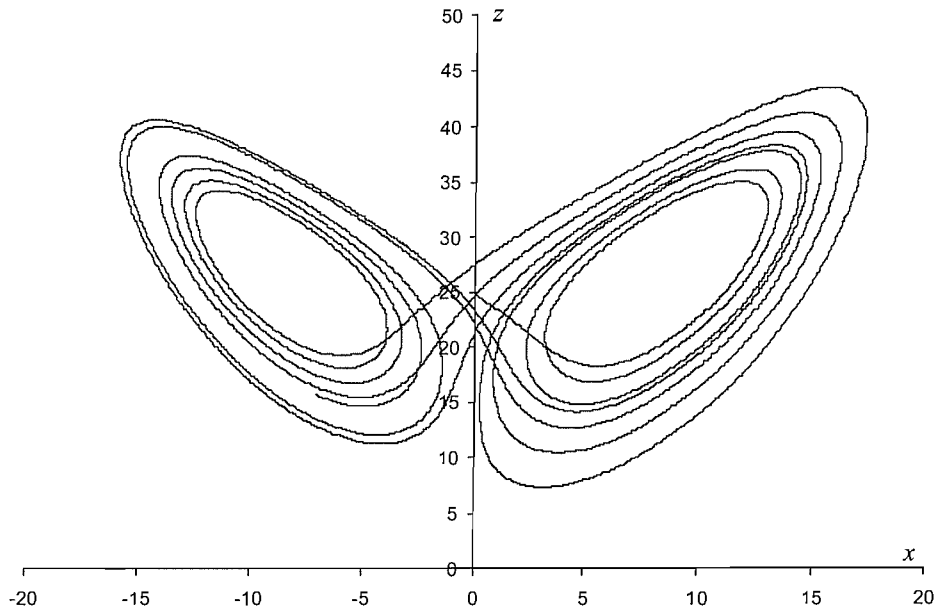

FIGURE 6.4: SystemC-A simulation of Lorenz chaos xz butterfly trajectory.

SystemC-A constructs, in particular, A/D and D/A interfaces and analogue circuit components.

The ideal boost SMPS consists of five basic components, namely a diode, a capacitor, an inductor, a power semiconductor switch, and a PWM controller. The SMPS uses a high frequency switch with varying duty cycle to maintain the output voltage. The output voltage variations caused by the switching are filtered out by a filter.

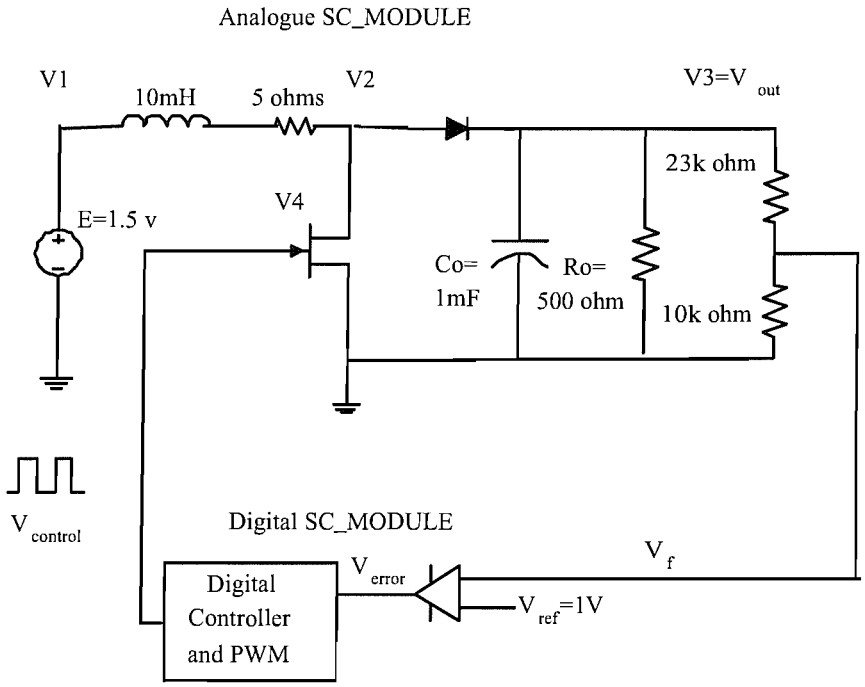

FIGURE 6.5: Boost 1.5V/3.3V switched mode power supply with digital control.

6.3.1 Modelling and Simulation

The SMPS SystemC-A model consists of two SystemC modules (SC_MODULE) and a testbench (higher module in hierarchy). One of the modules contains the analogue part of the system shown in Figure 6.5. The analogue module is modelled using circuit-level components from the simple analogue components library developed for SystemC-A, they have practical values. This example illustrates

how to construct a SPICE-like analogue circuit in SystemC-A as shown in Listing 6.4. The model defines circuit nodes (lines 2-6) and circuit components (lines 9-14), together with their parameters and connectivity. Listing 6.4 shows part of the analogue module, this part is contained in the module constructor since nodes and components should be instantiated only once prior to simulation.

```
1  //analogue module of SMPS
2   n0 = new sc_a_node("0");//creat nodes
3   n1 = new sc_a_node("n1");
4   n2 = new sc_a_node("n2");
5   n3 = new sc_a_node("n3");
6   n4 = new sc_a_node("n4");
7
8  //add components
9   sc_a_voltageS_dc *v1 = new sc_a_voltageS_dc("v1",n1,n0,1.5);
10  sc_a_inductor *l1    = new sc_a_inductor("l1",n1,n2,1e-2,5);
11  sc_a_diode *d1       = new sc_a_diode("d1",n2,n3,1,38.93,1e-13);
12  sc_a_capacitor *co   = new sc_a_capacitor("co",n3,n0,1e-3,1);
13  sc_a_resistor *ro    = new sc_a_resistor("ro",n3,n0,500);
14  sc_a_mosfet *M1      = new sc_a_mosfet("M1",n2,n4,n0,1,1,1e-8);
15  ...
```

LISTING 6.4: SystemC-A analogue module in the SMPS.

The PWM is modelled as a standard SC_MODULE at high level of abstraction, thus the SMPS contains two models at two different abstraction levels. Listing 6.5 shows the digital module where Listing 3.3 (page 60) shows the SMPS testbench with interfaces between analogue and digital modules. The DA interface (*interfaceDA*), placed between the PWM and the MOSFET transistor, smoothes the signal values propagating to the analogue solver since abrupt changes may cause problems in the analogue analysis (details of this idea are explained in Section 3.4).

The system was simulated for 0.2 seconds, when it reached the required voltage. Figure 6.6 shows the transient of the output voltage illustrating that the SMPS reaches steady state at about 0.2 seconds. Sample results at steady state are presented in Figure 6.7. The results showed the ripple in the output voltage waveform, error signal of the output voltage, inductor current, fast switching of $V_{control}$ and voltage at node number 2 (V_2). SMPS simulation was smooth and did not encounter any numerical difficulties. Simulation statistics are shown in Table

6.1. A simulation of 200m seconds undergo a CPU time of 232.1 seconds which means 2 million time points.

```
1  #include "systemc.h"
2  # define steps 100  //duty cycle divided into 100 steps
3
4  SC_MODULE(digital){
5      sc_in<double>Vd_in;// input analogue port
6      sc_out<bool>VcontD;// output digital port
7      sc_in<bool>clk;
8
9      double Verror_acc;
10     int D, tick_no;
11
12     void control(){   //process to control duty cycles
13     ...
14         double Gain2=10;// amplifier gain
15         Verror_acc+=1-(Vd_in.read()/3.33);//calculate error of output voltage
16
17         if (tick_no>steps){// this part will run once for each duty cycle
18             tick_no=0; //get back to right value
19             D=(Verror_acc/steps)*Gain2+66; //duty cycle
20             Verror_acc=0; //for the next cycle calculations
21
22             if (D<0)     //to limit the duty cycle 0-95%
23                 D=0;
24
25             if (D>0.95*steps)
26                 D=0.95*steps;
27
28         }//end if
29
30         if (tick_no<D) //produce output control signal
31             VcontD.write(true);
32         else
33             VcontD.write(false);
34
35         tick_no++;
36         if(tick_no>=steps)  //reset duty cycle
37             tick_no=0;
38     }
39     SC_CTOR(digital){
40         SC_METHOD(control);
41         sensitive << Vd_in;
42         sensitive << clk;
43         Verror_acc=100000;
44         tick_no=2*steps;
45     }
46 };
```

LISTING 6.5: SystemC-A PWM module in the SMPS.

TABLE 6.1: SMPS simulation statistics

Simulation time	200m seconds
Time step	0.1μ seconds
Number of steps	2 Millions
CPU time	232.1 seconds

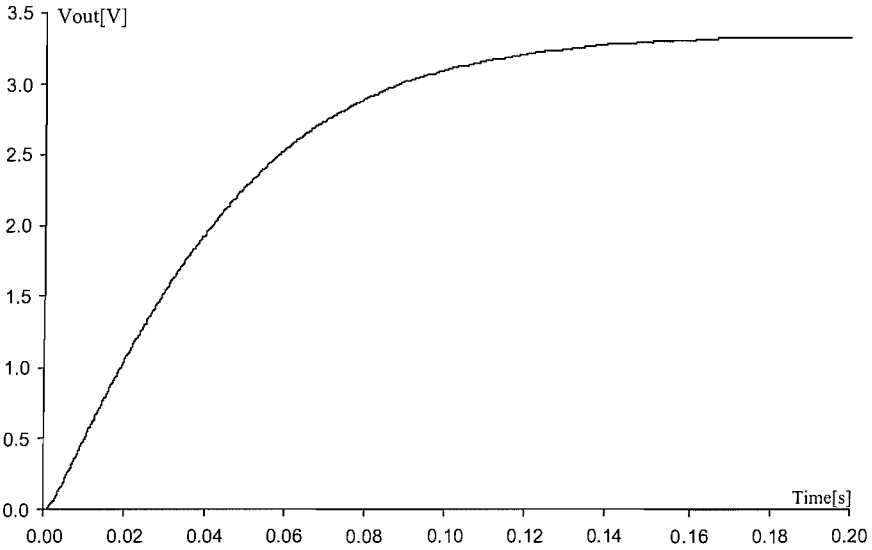

FIGURE 6.6: SystemC-A simulation of SMPS transition output voltage.

6.4 Phase Locked Loop

Phase locked Loops (PLL) [117, 118] are used in numerous applications, such as data communication, microprocessors, RF applications, and wireless systems. PLLs are used in these applications to implement a variety of functions, such as clock generation, frequency synthesis, clock recovery, and demodulation, where it is necessary to generate a precise signal frequency with low spurs and good phase noise. Figure 6.8 shows a block diagram of a basic PLL used as frequency synthesiser. Frequency synthesisers are used to produce digitally-controlled, stable, high frequency sources from a low frequency reference. It consists of a reference source, phase detector, charge pump, loop filter, Voltage Controlled Oscillator (VCO), and a divider. The filter and the VCO are analogue parts while others are digital parts. The divide ratio in this example is constant ($N = 2000$), hence the loop will operate to force the VCO signal frequency to be exactly N times that of the reference signal. The phase detector and charge pump output either positive or negative charge pulses depending on whether the reference signal phase leads or

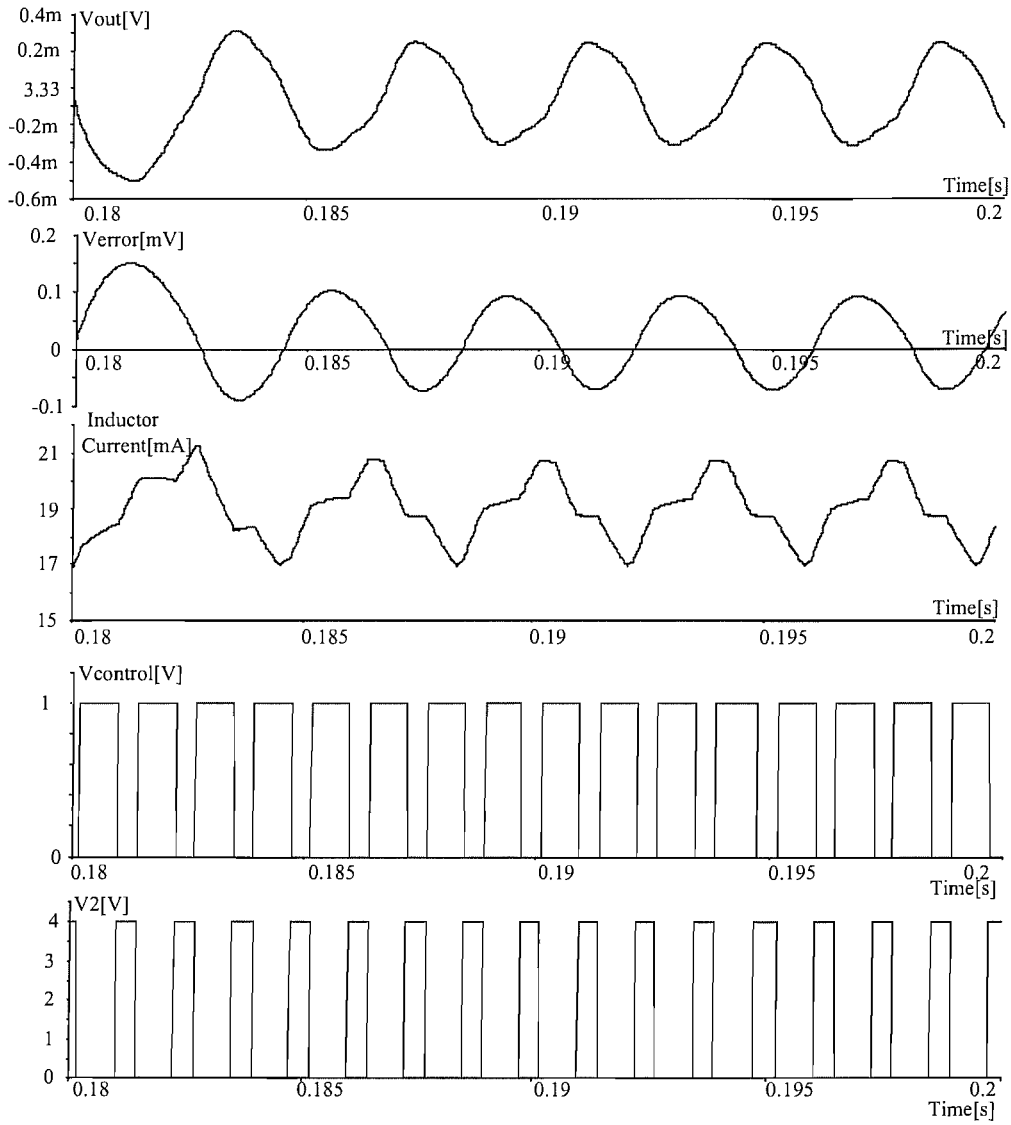

FIGURE 6.7: SMPS SystemC-A simulation results for a 200ms time window in steady state.

lags the divided VCO signal phase. These charge pulses are integrated by the loop filter to generate a tuning voltage.

One of the major concerns in the design of PLLs is noise or jitter performance [118]. The jitter from the PLL directly acts to degrade the noise floor and selectivity of a transceiver [119]. Noise sources in the system cause perturbations in the VCO control voltage resulting in variations in the output frequency.

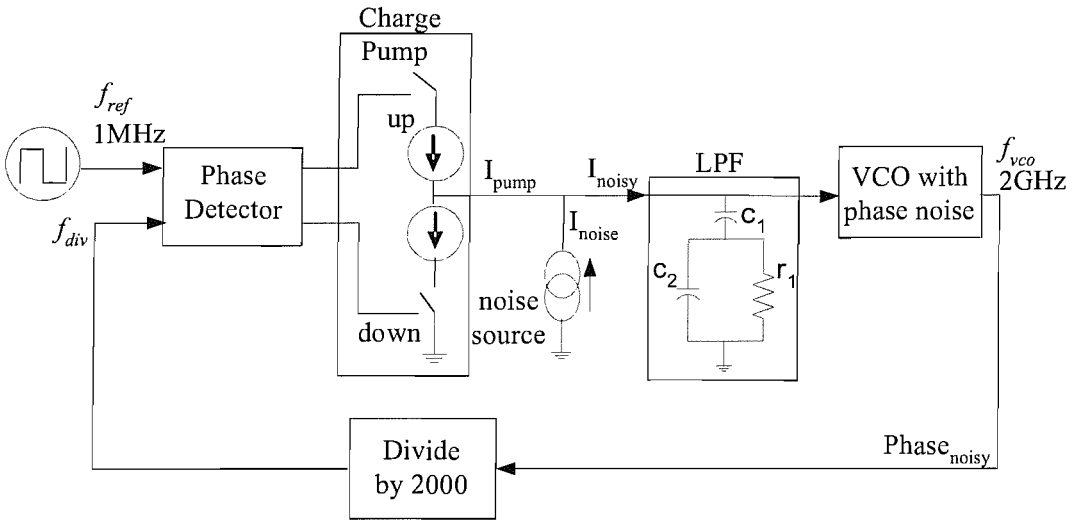

FIGURE 6.8: Block diagram of 2GHz Phase Locked Loop with noise and jitter.

6.4.1 Noise Module

PLL noise behaviour is difficult to predict with traditional circuit simulators because of the repetitive large-signal switching events, which are an essential part of the PLL operation, hence noise performance must be evaluated in the time-domain. Most classical simulators, SPICE being the best example, are not capable of simulating noise in PLLs as they can normally calculate small-signal noise around a quiescent operating point. Currently, the best suited simulator for PLL noise analysis is SpectreRF [119], which is capable of predicting the noise behaviour about a periodic operating point.

In SystemC-A suitable large-signal noise modules can be constructed with no difficulty. For this PLL example, a noise model is developed, it contains a standard function to generate a periodic process for a Gaussian white noise using the Box-Muller method. The function is to turn two uniformly distributed random sequences into two unity amplitude normal random X and Y (mean=0 and variance=1) sequences which can be scaled to the required levels. Listing 6.6 shows the noise module to produce charge pump current noise.

```
1  // generate samples from white noise source
2  // with a Gaussian normal amplitude distribution
3
4     double U1, U2, V1, V2, X. Y, r, Inoise;
5         do {
6              U1=rand()/3.2e4; //[0,1]
7              U2=rand()/3.2e4; //[0,1]
8              V1=2*U1-1;       //[-1,1]
9              V2=2*U2-1;       //[-1,1]
10             r = (V1*V1)+(V2*V2);
11        } while ((r == 0) || (r >= 1.0));
12
13    // transform into a normal distribution (Box-Muller transform)
14    X = V1 * sqrt(-2.0*log(r)/r)* 0.25;
15    Y = V2 * sqrt(-2.0*log(r)/r)* 0.25;
16
17    // return scaled sample
18    double scale=1e-6;
19    Inoise= scale * X;
```

LISTING 6.6: C++ noise model.

In this example two methods of modelling noise are implemented. Two different VCO models, presented in more detail in the following subsections, have been developed for both noise methods. The first method allows adding noise sources to analogue signals in any component and therefore provides a more accurate noise behaviour. The noise is injected by the controlled current source of the charge pump, although every PLL component is a potential noise source. The charge pump signal can be expressed as Eq.6.9.

$$I_{noisy}(t) = I_{cp}(t) + I_{noise}(t) \tag{6.9}$$

Consequently, the phase of the VCO is subject to noise (referred to as phase noise) which will manifest itself as jitter in the output waveform, Eq.6.10.

$$Phase_{noisy} = Phase(\theta + Jitter(\theta)) \tag{6.10}$$

In the first method the total effect of noise is modelled in the VCO by scaling and adding the generated noise model output to the VCO phase where it is turned into jitter. In the second noise method [120], a noise source is added to perturb the VCO pulse directly. This perturbation represents the total effect of any noise

source in the PLL. The second method, although cruder than the first one, has the advantage of shorter CPU times since the system will be simulated with larger step sizes to cover variations in the VCO pulses every 0.5n second.

6.4.2 VCO model (1)

The VCO generates a square wave whose frequency is proportional to the input signal level. In the VCO model the frequency is numerically integrated to compute the output phase which is used to generate the desired output signal. This is then followed by a modulus operation to keep the phase bounded, which prevents a loss of numerical precision that would otherwise occur when the phase becomes large after a long period of time. Output transitions are generated when the phase passes the value of 0.5 (the phase unit corresponds to a proportion of the duty cycle) in either direction. The VCO frequency is the rate of change of the phase,

$$\dot{\theta}(t) = \frac{d\theta}{dt} = f(v) = f_c + df * V_{filter} \tag{6.11}$$

where V_{filter} is the output voltage of the loop filter, f_c is the center frequency of the VCO, and $df = \frac{f_{max} - f_c}{V_{max}}$ is the VCO gain.

As SystemC-A allows different types of analogue descriptions to work together, the VCO was modelled here at behavioural level as a SystemC-A component described by an equation rather than a netlist at circuit level. The VCO is derived from the base *sc_a_component* class and contains build functions to add its contribution to the system Jacobian. Partial code of the VCO class is presented in Listing 6.7. Mixed OO-NQN equation formulation method is used to model the PLL, Listing 6.7 shows that VCO equation is formulated using pure Newton method (the use of *BuildM()* and *BuildB()* functions). The circuit components in the loop filter are formulated using Quasi-Newton method.

```
1  // VCO modelled as analogue component with one analogue node and
2  // one output boolean signal
3   vco::vco(char nameC[5],sc_a_system_variable *node_a, sc_signal<bool> *Vout):
4   sc_a_component(nameC,node_a, 0, value){
5       Vco=Vout;
6       theta = new sc_a_free_variable("theta"); //theta variable in VCO equation
7   }
8
9   void vco::BuildB(void){
10      ...
11      phase = X(theta);              // get the phase value from solution vector
12      phase = fmod(phase,1.0);       // limit the phase between 0.0-1.0
13      Pnoise = SampleNoise();        // generate scaled sample noise
14      PhaseNoisy = phase + Pnoise;// add noise to phase
15      if (PhaseNoisy > 0.5)          // produce VCO output
16          Vco->write(true);
17      if (PhaseNoisy < 0.5)
18          Vco->write(false);
19
20      //VCO parameters
21      fmix=0.5e9, fmax=5e9, Vmax=3.3, fc = 2e9;
22      df= (fmax-fc) / Vmax;          // calculate VCO gain
23      Qdotn = Xdot(theta);           // get the derivative
24      freq = fc + df * X(a);//tune VCO frequency using input voltage
25      if (freq < fmin ) //limit the frequency between fmin and fmax
26          freq = fmin;
27      else if (freq > fmin )
28          freq = fmax;
29
30      // main VCO equation
31      BuildRhs(theta,-Qdotn + freq);
32  }
33
34  void vco::BuildM(void){
35      fmin=0.5e9, fmax=5e9, Vmax=3.3, fc=2e9;
36      df= (fmax-fc) / Vmax;
37      S=S();          //get discretisation operator
38      freq = fc + df * X(a);
39      if (freq < fmin || freq> fmax)
40      {
41          if (freq < fmin )
42              freq = fmin;
43          else if (freq > fmin )
44              freq = fmax;
45          Jacobian(theta,theta,S);
46          Jacobian(theta,a,0);
47      }
48      else{
49          Jacobian(theta,theta,S);
50          Jacobian(theta,a,-df);
51      }
52  }
```

LISTING 6.7: SystemC-A VCO module using noise method (1).

The VCO interfaces are defined at line 3 of listing 6.7. The VCO is connected to the filter by a node (*sc_a_node*), while the output of the VCO is a SystemC signal of type boolean and connected to the divide by N module. The VCO equation (Eq.6.11) has one variable θ, which needs to be declared in the VCO constructor (line 6). A sample noise is generated at line 13 and added to the phase. The output is produced by evaluating the phase values (lines 15-18). The VCO equation RHS

is defined at line 31 and the jacobian elements are defined at lines 45-46 49-50, to integrate the frequency. The simulation results will be discussed after presenting the second noise method in the following section.

6.4.3 VCO model (2)

The second VCO model is a SystemC digital module, rather than an analogue component with a frequency integrator, as shown in Listing 6.8. This VCO model allows the second method of noise analysis. The method is to directly perturb the VCO output signal. The VCO model utilises SystemC *sc_event*. SystemC *sc_event* is used for process synchronisation. A process instant maybe triggered or resumed on the occurrence of an event (when the event is notified). In this example, *VCOphase* event is instantiated at line 6 and initialised at line 17. Process *Vf()* is sensitive to the *VCOphase* event. *Vf()* is functioning as follows: first it evaluates the time point and checks if time is proceeded to prevent multiple run at the same time. Then, the frequency is evaluated at line 28 using the input voltage *Vfilter*, consequently the phase is evaluated at line 34. A sample noise is generated at line 36 to represent jitter in VCO output signal. The jitter is added to *Tnext* variable to alter the time of the next pulse. The event is notified at perturbed phase periods using member function *notify(t,SC_SEC)* at line 41.

This noise method has an advantage of speed over the previous method since a larger step size is needed for simulations. A minimum of 0.2n seconds was used for this method while a 0.01n seconds was needed for the first noise method.

```
1  // VCO modelled as SC_MODULE
2  SC_MODULE(VCO2){
3       sc_in<double> Vfilter;    //input analogue port
4       sc_out<bool> Vout;        //output boolean port
5
6       sc_event VCOphase;        //instantiate an event on VCO signal
7       sc_signal<bool> Vosc;
8       void Vf();
9       double SampleNoise();
10      double freq, jitter, Tafter, period;
11      float Tnow, Tnext;
12
13      SC_CTOR(VCO2){
14           SC_METHOD(Vf);         //Vf process triggered by Ev_A event
15           dont_initialize();
16           sensitive << VCOphase;
17           VCOphase.notify(0, SC_SEC); //initialise the event
18           Vosc.write(0);
19           Tnext=0; jitter=0;
20           }
21 };
22
23 void VCO2::Vf(){
24      Tnow = sc_time_stamp().to_seconds();// get time
25      if (Tnow >= Tnext){                 // check if time proceeded
26           fmin=0.5e9, fmax=5e9, Vmax=3.3, df, fc=2e9;
27           df= (fmax-fc) / Vmax;
28           freq = fc + df * (Vfilter.read());
29           if (freq < fmin )
30                freq = fmin;
31           else
32                freq = fmax;
33
34           period=1/freq;
35           amp=25e-12;                  //noise magnitude
36           jitter = SampleNoise()*amp;  // scaled noise sample
37           Tafter = (period*0.5);
38           Tnext=Tnow+Tafter+jitter;    //alter VCO period
39           Vosc.write(!Vosc.read());
40      }
41      VCOphase.notify(Tnext-Tnow,SC_SEC);//calculations on VCO signal edges only
42      Vout.write(Vosc.read());
43 }
```

LISTING 6.8: SystemC-A VCO module using noise method (2).

6.4.4 Modelling and Simulation

With the first noise method, the system was simulated using extremely small analogue steps, much smaller than those calculated by the LTE control strategy. This was required to reflect accurately the effects of noise and jitter. The second noise method uses a simpler VCO model which does not require small step sizes as explained above. For the first noise method, to enforce a step size of 10ps or less, the charge pump module is sensitive to a 100GHz clock, whereas the digital modules are sensitive only to their input signals (see Figure 6.9 and Listing 6.9).

Figure 6.9 illustrates the PLL model in SystemC-A represented as block diagrams with detailed connectivity, whereas Listing 6.9 presents PLL SystemC-A testbench model.

Listing 6.9 is a standard SystemC *SC_MODULE* testbench which contains instantiations of the different blocks comprising the PLL. As is the case in any HDL, the module starts by defining global signals to provide connectivity to the instantiated modules, followed by instantiating the modules themselves. In Listing 6.9 *Clock1* enforces the simulation time steps by overriding the larger values calculated by the analogue stepping strategy. *Clock2* generates an input reference signal (*Reference*) with the frequency of 1MHz. The display module (*display1*) is represented as a *SC_CTHREAD* SystemC module to print the required signals at every clock pulse.

The module *interfaceAD* developed as part of SystemC-A, could have been used in this example to convert signals between the analogue and digital blocks. Instead, an alternative approach based on direct connection between the modules was used. The VCO and divide by N modules share the same digital signal and conversion between the analogue and digital parts is done implicitly within the VCO. Interfaces between modules can be implemented in many different ways, for example directly through signal ports, which is recommended especially at system level, or by nodes at analogue circuit level. In this example the connection between the LPF and VCO1 illustrates an analogue interface using node terminals. SystemC-A models of the phase detector, charge pump and filter, and Divide by N are provided in Appendix C.2.

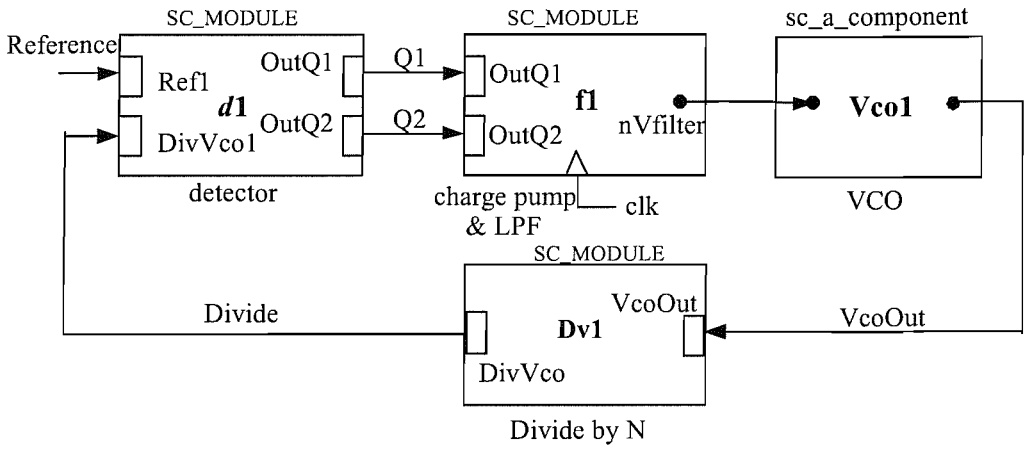

FIGURE 6.9: PLL model in SystemC-A represented as block diagram with detailed signals.

```
1  // PLL.h
2  ...
3  SC_MODULE(pll){
4      void Circuit();
5      sc_a_node *nVfilter;//instantiate a node to connect filter and VCO
6
7      SC_CTOR(pll){
8          Circuit();//circuit should be instantiated in constructor
9       }
10 };
11 // PLL.cpp
12  ...
13  void pll::Circuit(){
14     // clock for analogue parts
15     sc_clock CLOCK1("clock1", 0.2, SC_NS,0.5,0,SC_NS,true);
16
17     // clock to generate input reference signal
18     sc_clock CLOCK2("clock2", 1, SC_US,0.5,0,SC_US,false);
19
20     // instantiate signals to connect different nodules with each other
21     sc_signal<bool> Q1, Q2;
22     sc_signal<bool> Divide;
23     sc_signal<bool> Reference;
24     sc_signal<bool> Vco;
25     sc_signal<double> Iout;
26
27     input inp("input");// module to produce reference signal
28     inp.Ref(Reference);
29     inp.Clk(CLOCK1);
30     inp.Clk2(CLOCK2);
31
32     detector d1("detector");// detector module
33     d1.Ref1(Reference);
34     d1.DivVco1(Divide);
35     d1.OutQ1(Q1);
36     d1.OutQ2(Q2);
37
38     filter f1("filter");// charge pump and filter module
39     f1.OutQ1(Q1);
40     f1.OutQ2(Q2);
41     f1.clk(CLOCK1);
42     f1.Iout(Iout);
43     nVfilter=f1.n1 ;
44
45     DivideByN dv1("DivideByN");// divider module
46     dv1.Vco(Vco);
```

```
47      dv1.DivVco(Divide);
48
49      display1 displ("display1");// module to display all signals
50      displ.clk1(CLOCK1);
51      displ.Vco(Vco);
52      displ.Ref(Reference);
53      displ.Div(Divide);
54      displ.Q1(Q1);
55      displ.Q2(Q2);
56      displ.Iout(Iout);
57      displ.nVfilter=f1.n1;
58
59      vco *vco1=new vco("vco1", nVfilter, &Vco);// instantiate VCO component
60      sc_start(200,SC_US);//start simulation for 200u second
```

LISTING 6.9: SystemC-A PLL model.

The system response during the first eight micro seconds of the simulation, slow transients of the low pass filter voltage for both noise methods and histograms illustrating the VCO jitter are shown in Figure 6.10, Figure 6.11 and Figure 6.12 respectively. The histograms present the VCO jitter percentage occurrence for 5ps buckets and were calculated from the simulation results when the loop was in lock for both noise methods. Both sets of results illustrate similar behaviour.

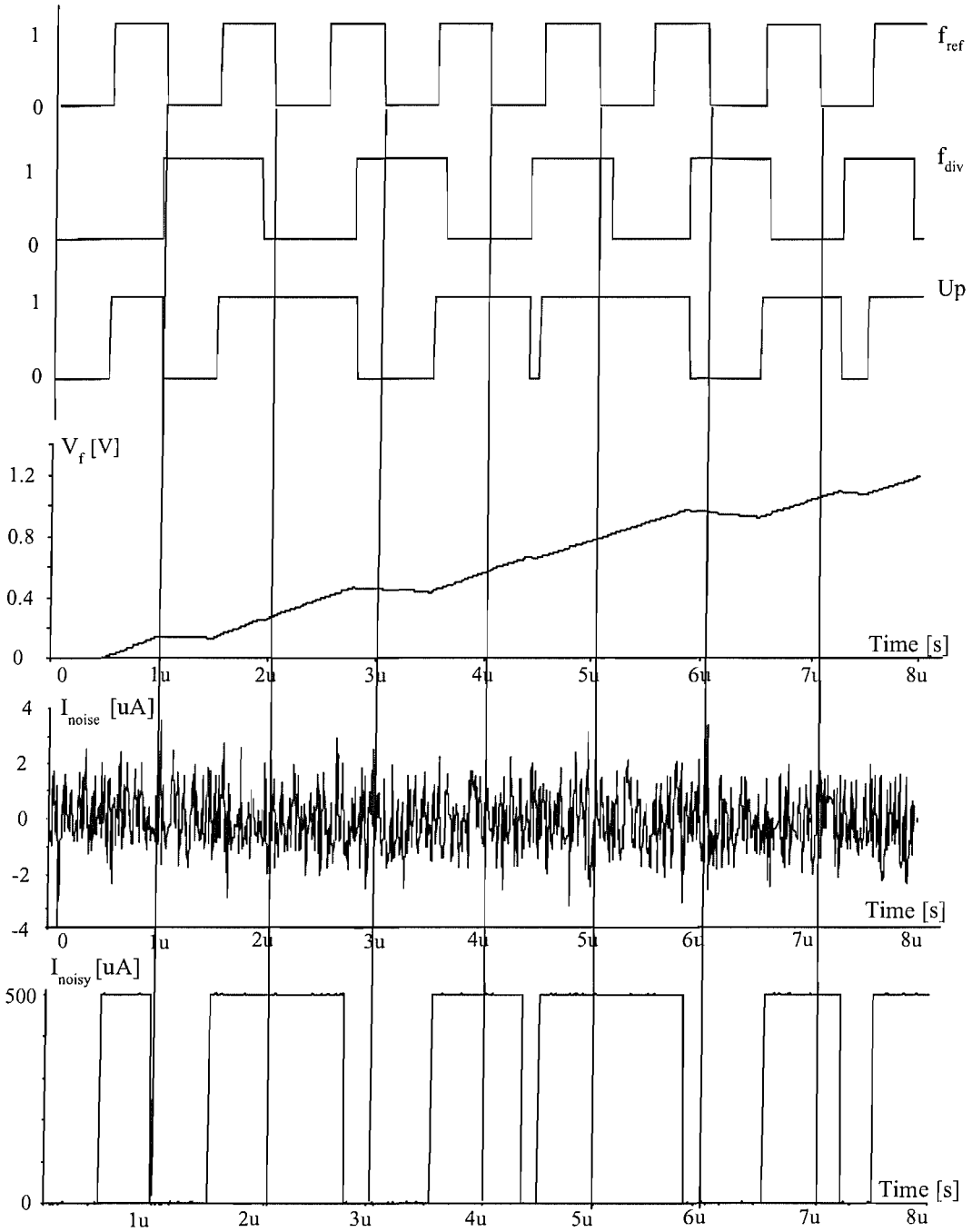

FIGURE 6.10: SystemC-A simulation results of the 2GHz PLL frequency synthesiser.

6.4.5 Comparison with VHDL-AMS

A comparison of analogue simulators is not necessarily a fair process because simulators vary in their algorithms, methods, accuracy criteria and many details are kept hidden. However, the developed PLL models have been used to compare

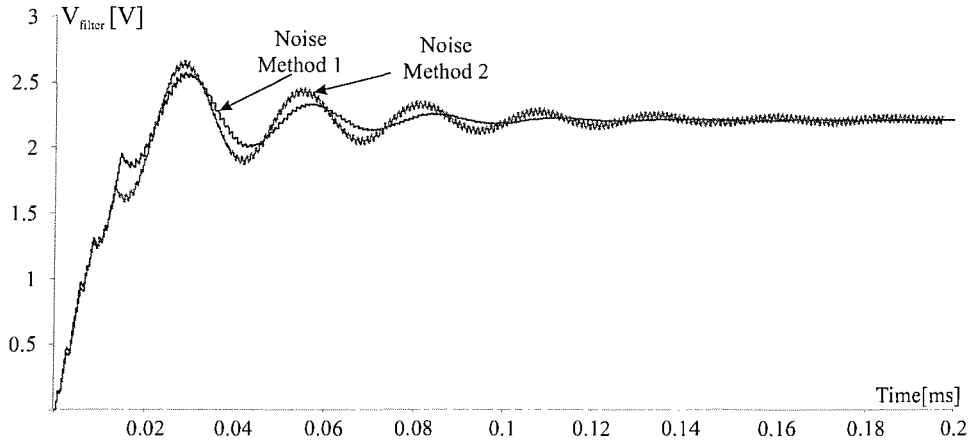

FIGURE 6.11: SystemC-A simulation of the low pass filter voltage for the two noise methods.

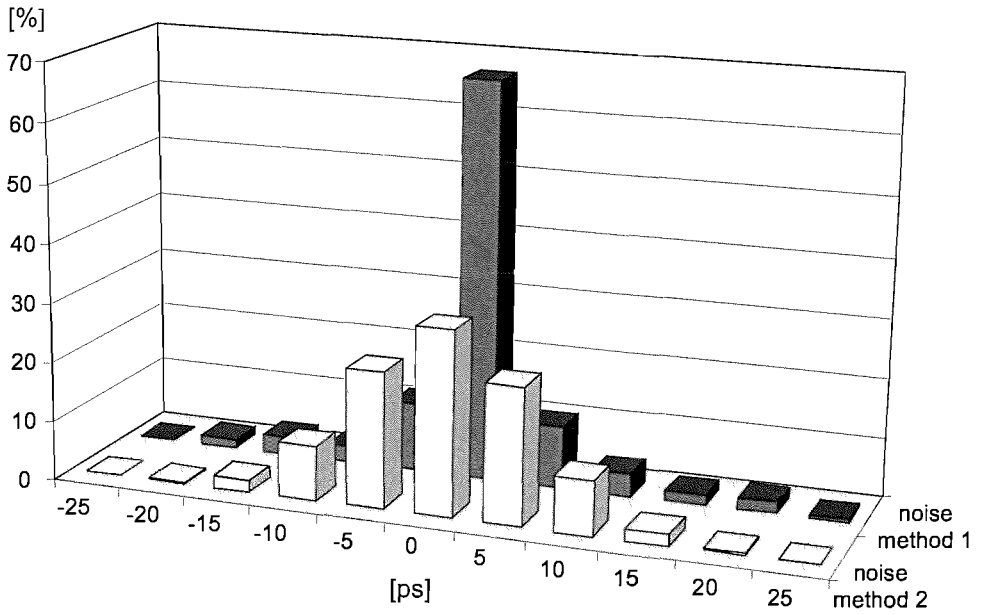

FIGURE 6.12: VCO jitter histogram for the two noise methods.

the speed of SystemC-A with that of SystemVision VHDL-AMS simulator from Mentor Graphics [113]. Simulations were carried out on a Windows 2000 computer with an AMD Athlon 1400 MHz processor and 512 MB RAM. A fixed time step was used in both simulators to suppress the effects of analogue time stepping factor. In the first noise method, where the 200μ second time interval was analysed with the time step of 0.01n second, SystemC-A took 16 minutes and 55 second while SystemVision took 1 hour and 4 minutes on the same machine. This represents a

factor of almost three times in favour of SystemC-A. The main reason behind the speed factor is the adoption of the efficient OO-NQN equation formulation method in SystemC-A, where this option is not provided in VHDL-AMS simulator.

For simulations with the second noise method, the system was simulated again for an interval of 200μ seconds and analysed with time step of 0.2n seconds. Simulations took only 80 seconds and 138 seconds in SystemC-A and SystemVision respectively. Table 6.2 shows relevant statistics.

TABLE 6.2: PLL simulation statistics.

	Noise Method (1)	Noise Method (2)
Number of steps	20 Millions	1 Million
Simulation time	200μs	200μs
Time step	0.01ns	0.2ns
SystemC-A CPU time	16m 55s	1m 20s
SystemVision CPU time	1h 4m 14s	2m 18s

6.5 Concluding Remarks

The chapter has demonstrated modelling and simulation four case studies to verify SystemC-A functionality from different aspects. The first two case studies were Van Der Pol oscillator (nonlinear system described by second order ODE) and Lorenz chaos (nonlinear system represented by 3 coupled ODEs) to demonstrate the ability of SystemC-A to model at behavioral level using its new language constructs. Modelling these simple examples was straight forward and was accomplished in short time with familiar syntax due to the good format provided by SystemC-A to model system equations.

The other two case studies were modelling SMPS and 2GHz PLL to test and demonstrate some of SystemC-A language elements such as A/D and D/A interfaces, analogue system variables and circuit-level electronic components. Furthermore, by modelling the last two case studies, SystemC-A has proved its capability

of handling highly complex systems which have disparate time scales of their transients and require excessive CPU times. Two efficient noise analysis techniques were developed when modelling the PLL. Finally, The examples demonstrate the capability of SystemC-A to model at different abstraction levels from system level down to circuit level.

Chapter 7

Electromagnetic System Modelling Case Study

For the purpose of validating SystemC-A developed constructs and methods, Chapter 6 modelled and simulated a suite of electrical systems which involves many difficulties. However, in this chapter, Ferromagnetic Hysteresis is modelled and simulated to illustrate the powerful SystemC-A capabilities to model non-electrical systems.

Nonlinear ferromagnetic components are used in many circuits and systems such as inductors and transformers. The widely used Jiles-Atherton (JA) model of ferromagnetic hysteresis [121, 122, 123] is adopted here. The JA model has been used extensively for creating non-linear models of magnetic materials for use in circuit simulation. The JA model is implemented in many commercial circuit simulators, such as SPICE [12] and SABER [124]. JA model is used in preference to other models because it is based on physical phenomena while other models usually employ look-up tables [125] and controlled sources to perform piecewise-linear approximation of different regions of the hysteresis curve. Look-up tables

are not practical since the tables have to be recalculated each time the parameters of the core material change.

Both VHDL-AMS and Verilog-AMS have been used to develop new models of ferromagnetic [126, 127, 128, 129]. Practical implementation of the JA model is not straightforward. It involves numerical integration of a discontinuous and non-linear differential equation. In addition, the model in its original form can sometimes produce a hysteresis curve with negative slopes which has no physical justification. Also, when simulating JA model, there were claims that the model suffers from convergence problems [130] and long analysis times [125].

The remainder of this chapter is organised as follows: In Section 7.1 a mathematical background of the JA model is briefly explained and a new Langevin's function is proposed for better numerical stability. The ferromagnetic hysteresis model is implemented in two different approaches. First, the model which involves using the simulator's analogue solver and suffers from numerical difficulties is implemented in Section 7.2 where the numerical difficulties are illustrated in Section 7.3. Then, a new model is presented in Section 7.4 which does not involve using the analogue solver and which overcomes most of the reported problems by using a special, timeless discretisation technique to integrate the magnetisation slope $\frac{dM}{dH}$. Unlike most existing implementations (e.g.[131, 132]), the new technique does not rely on time-based integration of $\frac{dM}{dH}$ and consequently does not involve the underlying analogue solver. Finally, Section 7.6 gives the conclusion.

7.1 Theory of Jiles-Atherton Model

This section describes the equations governing the JA model. In a saturable magnetic core, the relationship between its magnetic flux density B versus magnetic

field intensity H represents the shape of the magnetic hysteresis curve BH. An accurate implementation of the BH hysteresis curve is very important since several figures of merits can be drawn from it, for example the saturation level, remanence (or retentivity) and coercivity as shown in Figure 7.1. B and H are related by

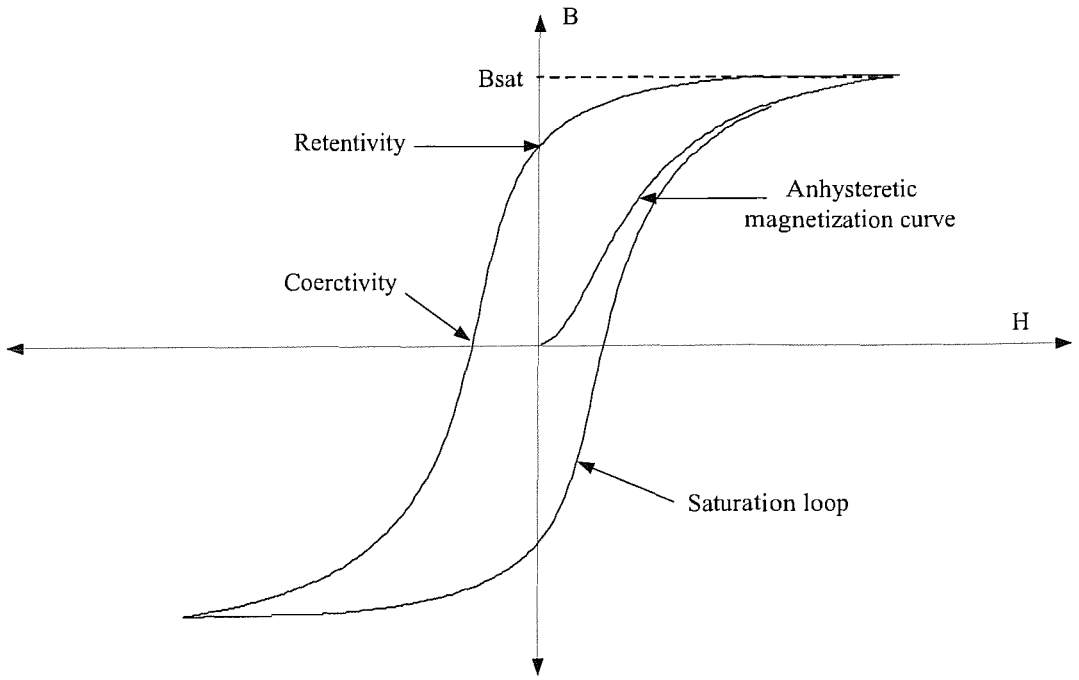

FIGURE 7.1: BH curve of magnetic hysteresis.

Eq.7.1 where $\mu_0 = 4\pi 10^{-7}$H/m is the permeability of free space and M is the total magnetisation within the material [122].

$$B = \mu_0(H + M) \tag{7.1}$$

The effective magnetic field intensity H_{eff} is defined by Eq.7.2 as the sum of the applied field H and some averaged contribution of the magnetisation M, where α is the average parameter of the magnetic field.

$$H_{eff} = H + \alpha M \tag{7.2}$$

If a magnetic material was able to return all of the magnetic energy that was input to it, the resulting magnetisation curve would take the form of a single valued sigmoid known as the anhysteretic magnetisation M_{an} curve. M_{an} as expressed in Eq.7.3 represents the lossless magnetisation of a material.

$$M_{an} = M_{sat} L(H_{eff}) \tag{7.3}$$

where M_{sat} is the saturation level of M. $L(H_{eff})$ is usually expressed by the well-known Langevin's function in Eq.7.4, where a is shaping coefficient to adjust the curve according to the magnetic hardness of the material.

$$L(H_{eff}) = \coth(\frac{H_{eff}}{a}) - \frac{a}{H_{eff}} \tag{7.4}$$

This Langevin's function can become numerically singular for small values of the magnetic field H_{eff}. It is standard practice to implement a simple approximation for small values of H_{eff} as shown below:

$$L(H_{eff}) = \coth(\frac{H_{eff}}{a}) - \frac{a}{H_{eff}} \quad for \quad |H_{eff}| > 10^{-3} \tag{7.5}$$

$$L(H_{eff}) = \frac{H_{eff}}{2a} \quad for \quad |H_{eff}| \leq 10^{-3} \tag{7.6}$$

In this research a different approximation of Langevin's function is used (Eq.7.7) to avoid the numerical singularity.

$$M_{an} = M_{sat} \frac{2}{\pi} \tan^{-1}(\frac{H_{eff}}{a_2}) \tag{7.7}$$

where a_2 has the same meaning as a but with a different value. Normalised Langevin's function and the approximation in Eq.7.7 of the anhysteretic function are compared in Figure 7.2 and show a very similar behaviour. The advantages of the new approximation are simpler implementation and continuity through zero.

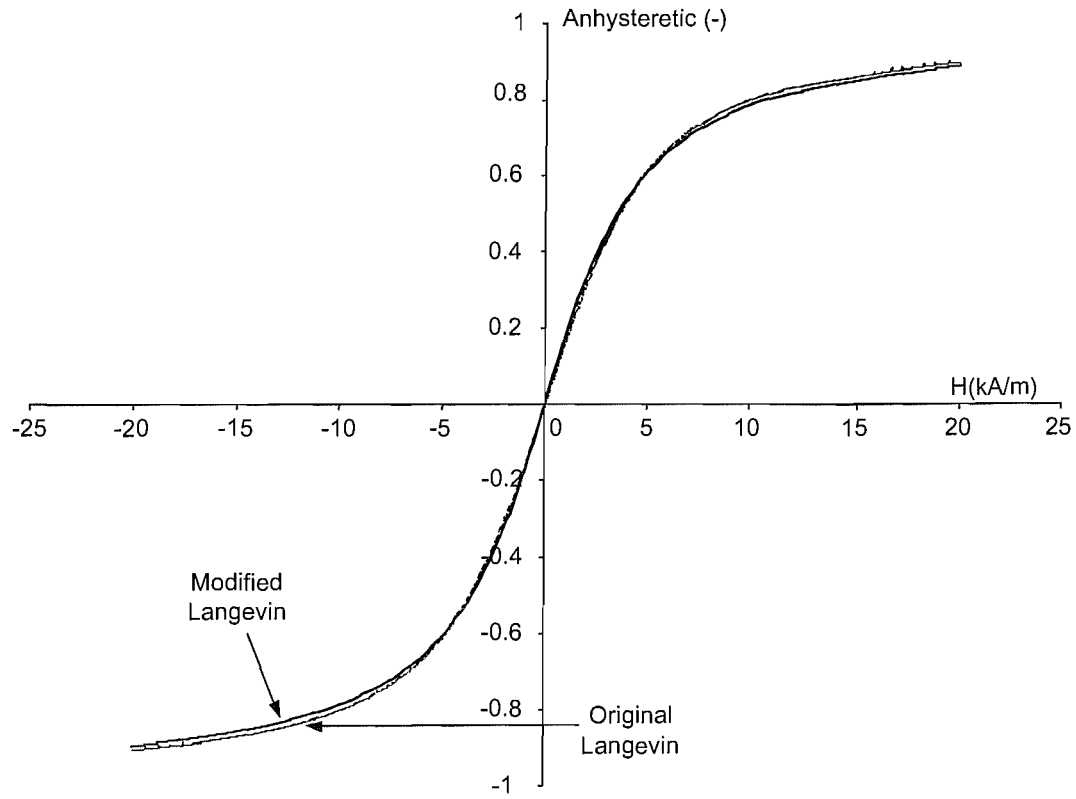

FIGURE 7.2: Original and modified anhysteretic functions.

The total magnetic field M in a ferromagnetic material is composed of the reversible M_{rev} and irreversible M_{irr} components (Eq.7.8). M_{irr} represents the energy dissipated while the material is magnetised.

$$M = M_{rev} + M_{irr} \tag{7.8}$$

In the JA model these components are related by Eq.7.9.

$$M_{rev} = c(M_{an} - M) \tag{7.9}$$

where c is the domain wall flexing constant. Substituting Eq.7.9 in Eq.7.8 results in Eq.7.10:

$$M = \frac{1}{1+c} M_{irr} + \frac{c}{1+c} M_{an} \tag{7.10}$$

The rate of change of the M_{irr} is proportional to the distance of the total magnetisation to M_{an} as expressed by Eq.7.11:

$$\frac{dM_{irr}}{dH} = \frac{M_{an} - M}{\frac{\delta k}{\mu_0} + \alpha(M_{an} - M)} \tag{7.11}$$

where δ is the sign of $\frac{dH}{dt}$ and k is a material dependent variation which gives a measure of the hysteresis loop width. Finally, the total differential equation of hysteresis (Eq.7.12) is obtained by substituting Eq.7.11 in Eq.7.10 and differentiating both sides.

$$\frac{dM}{dH} = \frac{1}{(1+c)} \frac{M_{an} - M}{\frac{\delta k}{\mu_0} + \alpha(M_{an} - M)} + \frac{c}{(1+c)} \frac{dM_{an}}{dH} \tag{7.12}$$

Other variations of the JA model are outlined in their series of papers [121, 122, 123]. The most important challenge in the implementation of the JA model is the calculation of the magnetisation slope given by Eq.7.12. Eq.7.12 is a nonlinear differential equation with incremental terms which needs to be solved. Most implementations of the JA model require conversion of the magnetisation derivative $\frac{dM}{dH}$ to time derivatives. This is usually implemented by calculating the derivative of H with respect to time $\frac{dH}{dt}$, and then integrating $\frac{dM}{dt}$ (e.g. using the VHDL-AMS 'INTEG operator).

For the proposed model in Section 7.4, the integration is implemented using two integration methods, Forward Euler and 4^{th} order Runge-Kutta. The Forward Euler integration method to solve $\frac{dy}{dx} = f(x, y)$ has the following finite difference form:

$$y_{i+1} = y_i + hf(x, y) \tag{7.13}$$

where y_i for $i = 0, 1, \ldots$ are the calculated solution points at x_i, h is the time step. The 4^{th} order Runge-Kutta method is described by the following formulas:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (7.14)$$

$$k_1 = hf(x_i, y_i) \quad (7.15)$$

$$k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}) \quad (7.16)$$

$$k_3 = hf(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}) \quad (7.17)$$

$$k_4 = hf(x_{i+1}, y_i + k_3) \quad (7.18)$$

where k_1, k_2, k_3, and k_4 are coefficients.

Although the latter method is more accurate, the simple Forward Euler method was also tried for comparison as the step size h is limited by factors other than the truncation error of the discretisation.

An important feature of a JA model implementation is its treatment of minor loops. When minor loops oscillate between two values H_{min} and H_{max}, they gradually drift towards an equilibrium loop. This phenomenon is known as accommodation [133]. A good model should be capable of producing minor loops with no numerical difficulties for various minor loops sizes and in different positions.

7.2 Modelling and Simulation of the Original Jiles-Atherton Model

The JA model was first created using the original equations with the commonly used time integration. The JA model is modelled in SystemC-A as a component class connected to a sine wave voltage source via two nodes. Components and nodes are instantiated in a testbench as in listing 7.1.

```
1  // instantiate magnetic nodes
2   p1 = new sc_a_node("p1");
3   m1 = new sc_a_node("0");
4
5  // instantiate a sin wave voltage source and the JA model
6   sc_a_voltageS_sin *I1 = new sc_a_voltageS_sin("Vsin",p1,m1,
7                                    0.16,0,10000,0,0);
8                                //frequency, offset, amplitude, delay, damping
9
10  JA *JA1 = new JA("JA1",p1,m1);
11  JA1->generic(4000, 0.1, 1.6e6, 0.003, 2000, 1, 1, 4e-6);
12             //(k, c, ms, alpha, a, ur, length, area)
13  sc_start(20,SC_SEC);// start simulation for 20 seconds
```

LISTING 7.1: SystemC-A testbench for the Jiles-Atherton simulation.

The model is simulated using the parameters shown in Table 7.1. Their values are identical to those used in the original paper by Jiles and Atherton [121] except for $a2$ which is used in Section 7.4 in modelling the modified JA.

Symbol	Definition	Value
k	pinning parameter of the domain wall (A/m)	4000
c	domain wall reversible movement parameter	0.1
M_{sat}	magnetic saturation	1.6M
α	averaging parameter of the magnetic field	0.003
a_1	parameter of the original anhysteresis curve shape	2000
a_2	parameter of the modified anhysteresis curve shape	3500

TABLE 7.1: Jiles-Atherton model parameters.

Listing 7.2 shows the JA component model. The JA constructor is defined in (lines 18-26). The model has four SystemC-A system variables, two is of type *sc_a_node* to describe the model connection nodes, and the other two are of type *sc_a_free_variable* to perform differentiation and integration involved in the JA model. A differentiator operator is used in line 32 to differentiate H_{eff}, whereas an integration operator is used in line 55 to integrate $\frac{dM}{dt}$. *JA::generic()* (lines 72-75) is a function through which the user can provide the system parameters to the model from the testbench. *JA::BuildB()* (lines 28-61) implements the common way of solving the hysteresis equation by multiplying the time derivative of H by $\frac{dM}{dH}$ and then integrate $\frac{dM}{dt}$. Langiven's function represented by Eq.7.4 (*JA::Lang()*) is implemented at lines 63-70.

```
1  class JA:public sc_a_component{
2  public:
3      JA();
4      JA(char nameC[5],sc_a_system_variable *node_a,sc_a_system_variable *node_b);
5      virtual ~JA();
6      void BuildB(void);
7      double Lang(double x);
8      void generic(double k1,double c1,double ms1,double alpha1,double a11,double ur1,
9          double len1,double area1);
10     sc_a_system_variable *dMirrdt, HeQ;
11
12 protected:
13     double MU0, mg, h, he, B,man,mrev,mirr,mtotal,delta;
14     double k, c, ms, alpha, a1, ur, len, area, flux;
15     double dhdt,dm,dmdh,dmdh1,dmirrdt,mirrcalc,dMdt;
16 };
17  // JA constructor
18  JA::JA(char nameC[5],sc_a_system_variable*node_a,sc_a_system_variable*node_b):
19   component(nameC,node_a, node_b,0){
20     dMirrdt = new sc_a_free_variable("dMirrdt");
21     HeQ = new sc_a_free_variable("HeQ");
22     // initialise all variables
23     MU0=4e-7*PI, mg=0, h=0,he=0,b=0,man=0,mrev=0,mtotal=0,delta=0;
24     mirr=0, dhdt=0,dm=0,dmdh=0,dmdh1=0,dmirrdt=0,mirrcalc=0;
25     k=0, c=0, ms=0, alpha=0, a1=0, ur=0, len=0, area=0;
26  }
27
28 void JA::BuildB(){
29     h = X(a);
30     //calculate H effective and its derivative
31     he = h + (alpha *ms * mtotal);
32     dhdt=Xdot(HeQ,he);// Xdot(Systemvariable, value)
33    // Get the field direction
34     if (dhdt > 0.0)
35         delta = 1.0;
36     else
37         delta = -1.0;
38
39     // calculate anhysteretic and reverse Magnetisation
40     man = Lang(he/a1);
41     mrev = c * man / ( 1.0 + c );
42
43     //calculate incremental Magnetisation
44     dm=man-mtotal;
45
46     // calculate dMirrdH
47     dmdh1=dm/(delta*k-alpha*ms*dm);
48     if (dmdh1 > 0)//limit dmdh to positive values only
49         dmdh=dmdh1;
50     else
51         dmdh=0;
52
53     // calculate dM/dH and then integrate to get Mirr
54     dMdt=dhdt*dmdh;
55     mirrcalc= Xinteg(dMirrdt,dMdt);// Xinteg(Systemvariable, value)
56     mirr=1.0*mirrcalc/(1.0+c);
57     // Calculate Total Magnetisation
58     mtotal = mrev + mirr;
59     // Calculate Flux Density
60     B=MU0*(ms*mtotal+h);
61 }
62
63 double JA::Lang(double x){ //Langevin's function
64     double lang_x;
65     if (fabs(x) < 1.0e-3)
66         lang_x = 0.333 * x;
67     else
68         lang_x = 1/tanh(x) - 1.0/x;
69     return lang_x;
70 }
71
```

```
72  void JA::generic(double k1, double c1, double ms1, double alpha1,
73  double a1, double ur1, double len1, double area1){
74      k=k1;c=c1;ms=ms1;alpha=alpha1;a=a11;ur=ur1;len=len1;area=area1;
75  }
```

LISTING 7.2: SystemC-A implementation of the original Jiles-Atherton ferromagnetic hysteresis model.

A VHDL-AMS model [127] is used in order to compare the results, Appendix D.2 presents the code. The SystemC-A and VHDL-AMS models have the tendency of crashing at hysteresis cusps, i.e. at points where an abrupt change in the magnetisation slope occurs, despite trying various excitations, with various periods. For example, VHDL-AMS simulator produced the following errors:

```
TIME 1.543564e+000: end of non-convergence points
Newton: No convergence at time :1.543564E+009Nano ; try to pass over...

TIME 1.543564e+000: end of non-convergence points
Newton: No convergence at time :4.629691E+009Nano ; try to pass over...

TIME 4.629691e+000: end of non-convergence points
```

Figure 7.3 shows the input H waveform and output B for non-crashing state, while Figure 7.4 and Figure 7.5 shows the hysteresis curve resulted from the SystemC-A and VHDL-AMS simulations respectively.

The model was simulated for more than 20 seconds when it crashed during simulation. SystemC-A hysteresis curve was compared to the VHDL-AMS curve indicating the same behaviour of crashing.

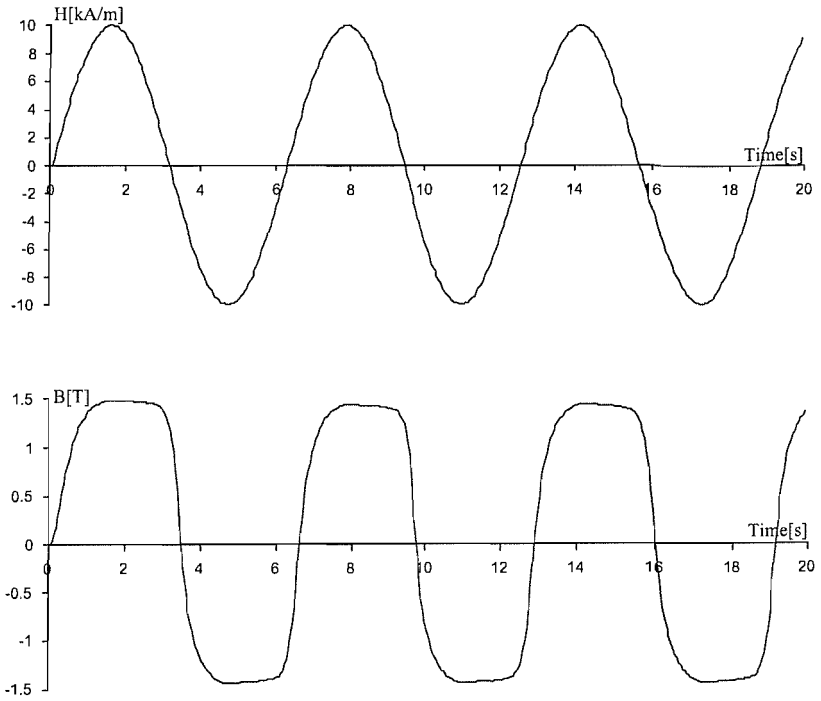

FIGURE 7.3: Sinusoidal B and H waveforms of ferromagnetic hysteresis simulation in SystemC-A.

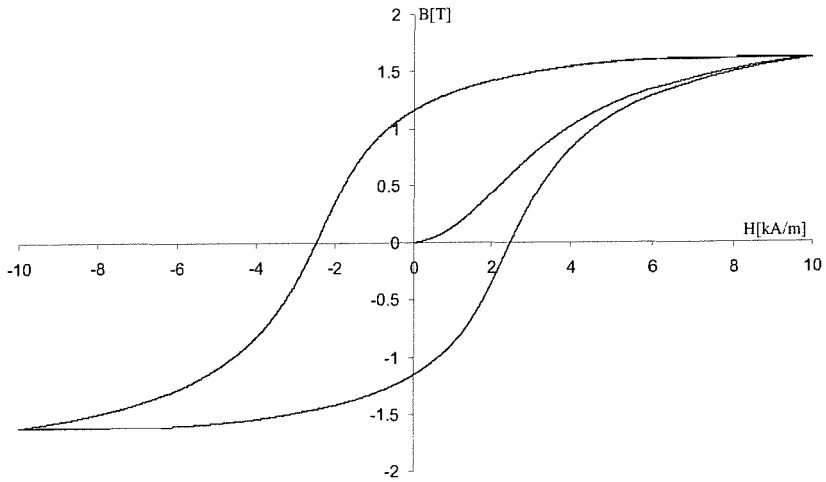

FIGURE 7.4: SystemC-A simulation of BH curve of the original JA model.

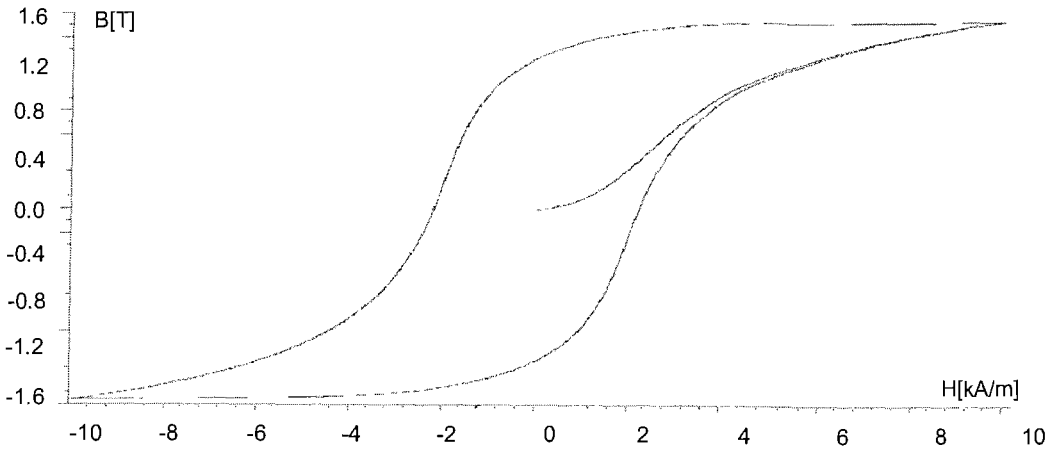

FIGURE 7.5: VHDL-AMS simulation of BH curve of the original JA model.

7.3 Nonphysical Behaviour and Numerical Difficulties of Jiles-Atherton Model

One of the well known shortcomings of the JA model is that it sometimes shows negative values of the magnetisation slope $\frac{dM}{dH}$ at a cusp of the loop [134]. This behaviour is nonphysical. At a cusp, $\frac{dM}{dH}$ can change abruptly and at that point $\frac{dH}{dt}$ must be zero because H is changing direction. As analogue simulation is a numerical march-in-time process using a sequence of discrete time points rather than continuous time, derivatives are estimated with finite-difference expressions. Consequently, it is possible that the calculated values of $\frac{dM}{dH}$ might be non-zero at a cusp, Figure 7.6 presents this numerical phenomenon. Figure 7.6 illustrates abrupt changes of $\frac{dM}{dH}$ on passing a cusp as a result from numerical approximation. The figure shows that when H is changing direction, $\frac{dM}{dH}$ decreases abruptly to a negative value. Part (C) of the figure shows the rectified derivative to remove this nonphysical behaviour.

Not only does this model behaviour exhibit nonphysical results, but it also presents substantial computational difficulties as the slope $\frac{dM}{dH}$ passes through zero. Analogue simulators usually handle abrupt changes in the solution by backtracking

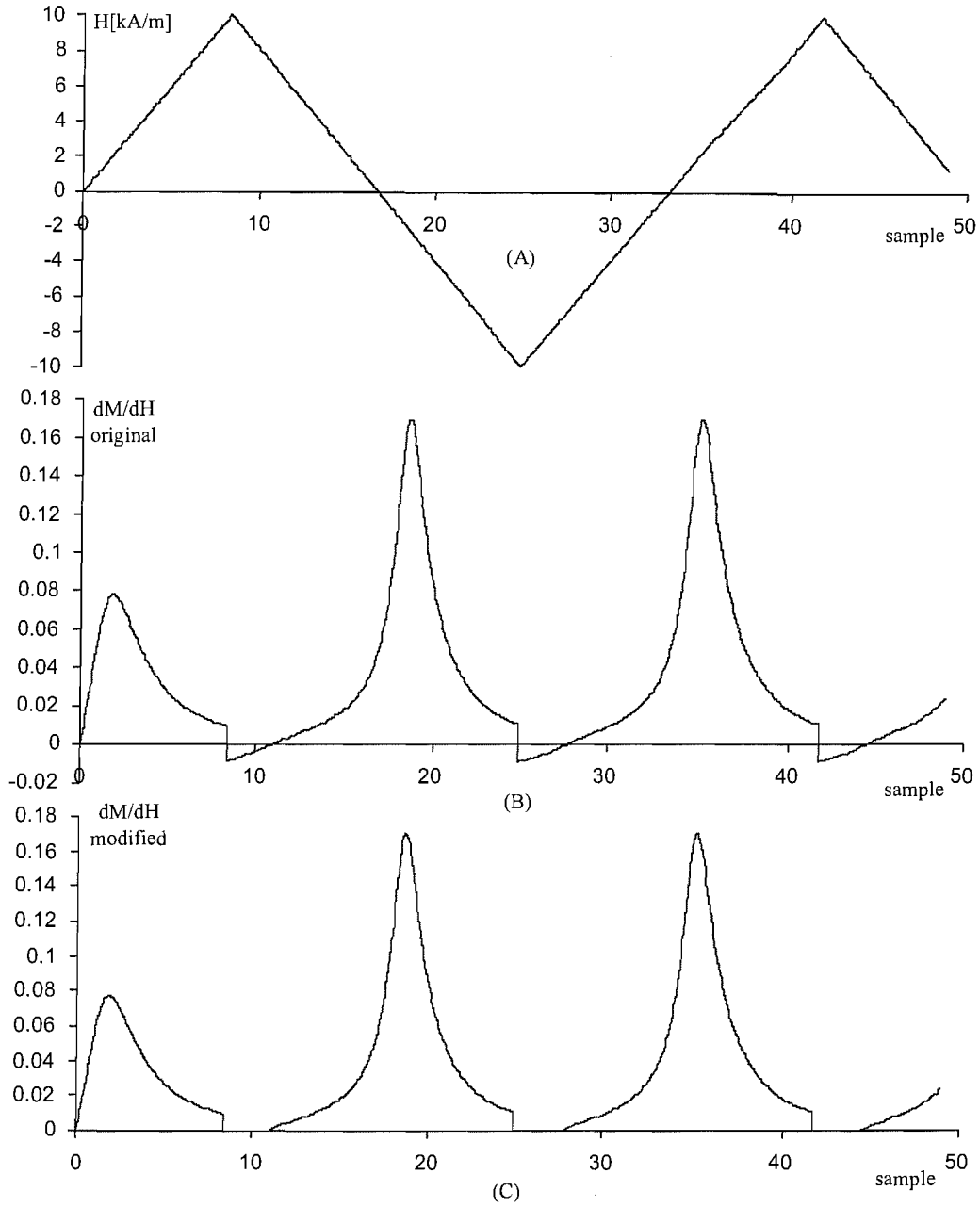

FIGURE 7.6: Original and modified $\frac{dM}{dH}$ resulting from a DC sweep of H.

and recalculating the solution with smaller time steps. This approach, however, is likely to fail, as has been reported [134], when dealing with the inherently discontinuous magnetisation slope of the JA model. The modified model presented in the next section assures that the derivative $\frac{dM}{dH}$ is positive after reversal. Further, the use of explicit integration in a solution process controlled by the model, has

proved an effective way of avoiding convergence problems at the slope discontinuity points.

7.4 Modelling and Simulation of the Modified Jiles-Atherton Model

To overcome the difficulties in Section 7.3, this section presents a new model implementation of the JA model. The new model is based on timeless discretisation technique to integrate the magnetisation slope $\frac{dM}{dH}$. The new method is more general as it does not depend on the simulator's underlying analogue solver and uses an independent process. The process is responsible for directly performing timeless integration of $\frac{dM}{dH}$ using H as the independent variable, not the time. The new JA model has been implemented in SystemC as digital *SC_MODULE* as shown in Listing 7.3.

```
1  #include "systemc.h" // header file
2
3  SC_MODULE(JA){
4      sc_in<double> H;
5      sc_out<double> Msig, Bsig;
6      sc_signal<bool> hchanged, trig;
7      sc_signal<double> deltah, lasth ;
8
9      void core();
10     double Lang_mod(double x);
11     void monitorH();
12     void Integral();
13     double MU0,mg,dhmax,He,B,man,mrev,mirr,mtotal;
14     double k,c,ms,alpha,a,area,mirr,flux;
15     void generic(double k1, double c1, double ms1,
16                  double alpha1, double a1, double area1);
17
18     SC_CTOR(JA){
19         SC_METHOD(core);
20         sensitive << H;
21
22         SC_METHOD(monitorH);
23         sensitive_pos << hchanged;
24
25         SC_METHOD(Integral);
26         sensitive << trig;
27      //initialise all variable
28      MU0=4e-7*PI,mg=0,dhmax=12,He=0,B=0,man=0,mrev=0,mtotal=0,
29      lasth=0,deltah=0,a=0,mirr=0,k=0,c=0,ms=0,alpha=0,area=0;
30      }
31   };
32   //CPP file
33   #include "JA.h"
```

```
34
35   void JA::core(){
36     // hchanged signal triggered by sufficient changes in field strength
37     if (fabs(H - lasth) >  dhmax)
38         hchanged = 1;
39
40     He = H + (alpha * ms * mtotal); //calculate effective field
41     man = Lang_mod(He/a);//calculates anhysterestic from modified Langevin's function
42     mrev = c * man / ( 1.0 + c ); //reversible magnetisation
43     mtotal = mrev + mirr; //total magnetisation
44     B=MU0*(ms*mtotal+H);
45     Msig.write(mtotal);
46     Bsig.write(B);
47   }
48   double JA::Lang_mod(double x){ //modified Langevin's function
49     double lang_x;
50         lang_x = (2/3.14159265)*atan(x);
51     return lang_x;
52   }
53  void JA::generic(double k1, double c1, double ms1,double alpha1,
54   double a1,double area1){k=k1;c=c1;ms=ms1;alpha=alpha1;a=a1;area=area1;}
55
56  void JA::monitorH(){//monitorH() is triggered by hchanged
57     double dh;
58     dh=H - lasth;//calculate dh
59     if (fabs(dh) > dhmax){
60         deltah=dh;
61         lasth = H;
62         trig=1;//trigger Integral()
63         hchanged=0;}
64   }
65   void JA::Integral(){//triggered by trig signal and perform integration
66     double deltam, dm, dmdh, dmdh1, dh, dk;
67      // Get the field direction
68      if (deltah > 0)
69          dk = k;//rising
70      else
71          dk = -k;//falling
72
73      // Forward Euler integration method
74      dh=deltah;
75      deltam = man - mtotal;
76      dmdh1 = deltam/((1+c)*(dk - (alpha*ms*deltam)));
77      if (dmdh1 > 0.0)// to assure positive derivatives
78          dmdh = dmdh1;
79      else
80          dmdh = 0.0;
81     dm = dh * dmdh;
82      if (dm * dh < 0.0)
83          dm = 0.0;
84
85    //JA model
86    mirr = mirr + dm ;
87   }
88
```

LISTING 7.3: SystemC-A implementation of the proposed Jiles-Atherton ferromagnetic hysteresis model.

The new JA model in Listing 7.3 is connected through ports to another module which supplies input waveforms. The main model body process *JA::core()* at line 35 is triggered on changes of the external magnetic field H. It calculates the anhysterestic from the modified Langevin's function according to Eq.7.7

(*JA::Lang_mod()*) as well as the reversible and total magnetisation. *JA::core()* triggers the processes *JA::monitorH()* and *JA::Integral()* if an update of the magnetisation slope is necessary. The code in Listing 7.3 shows the Forward Euler implementation of the magnetisation slope integral and the Runge-Kutta version was implemented in a similar way. The timeless approach to the slope discretisation avoids using time as the independent variable and the integral is calculated using increments of the magnetic field H rather than time steps.

Simulations of the new model in Listing 7.3 were working for all input values of H and never crashed at cusp reversals which indicates a numerical stability of the timeless magnetisation slope integration. To illustrate how the model handles asymmetric and non-symmetric minor loop behaviour, testbenches were developed to simulate sequences of non-biased (i.e. symmetric) minor loops and asymmetric minor loops biased with a field value of 2kA/m. Figure 7.7 shows applied field strength excitations H together with flux density outputs B. For generality, a triangular waveform is used in a DC sweep, i.e. timeless simulations. The corresponding BH curves with non-biased and biased minor loops are shown in Figure 7.8.

The results clearly demonstrate the property of long-term magnetisation memory loss in the JA model. As the JA model gradually loses its long term magnetisation memory, minor loops do not exhibit a closure after one cycle but eventually converge to a closed loop as shown in Figure 7.8. While these experiments indicate the correctness and numerical reliability of the timeless discretisation technique for the magnetisation slope, this strange behaviour of minor loop sequences in the JA model has not been confirmed experimentally [135].

Simulation results showing major and minor loop behaviour are consistent with other reported implementations using different languages and simulation tools [131, 136]. In this context, it is worth pointing out that other, non-JA models

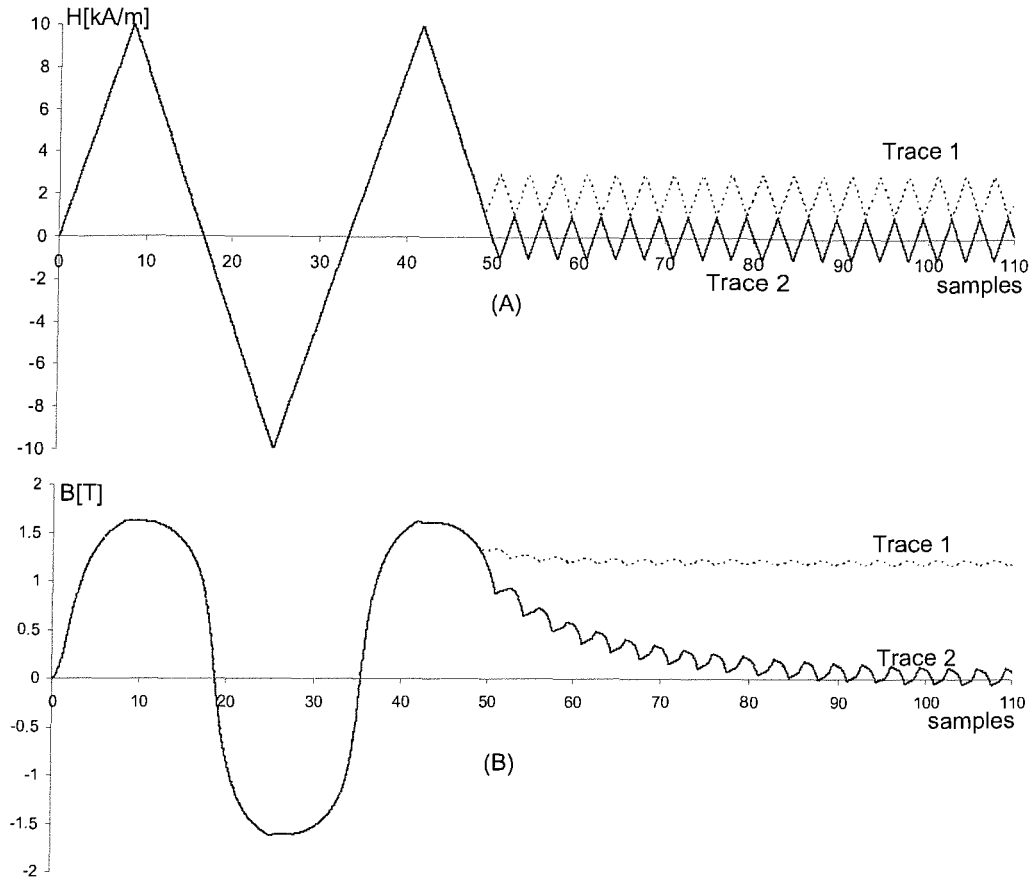

FIGURE 7.7: DC sweep simulations of the SystemC model showing the excitation H and response B. Trace 1 includes minor loops biased at $H = 2\text{kA/m}$ and trace 2 - non-biased minor loops.

may exhibit different behaviour [137], specifically the Preisach model [138] which tends to produce different results for minor loop behaviour around turning points.

7.5 Comparison with VHDL-AMS

SystemC-A is capable of modelling non-electrical domain systems using the new language constructs and methods. Also, by being HDL based on C++, JA model was reformed into a new model to overcome most of the reported problem easily.

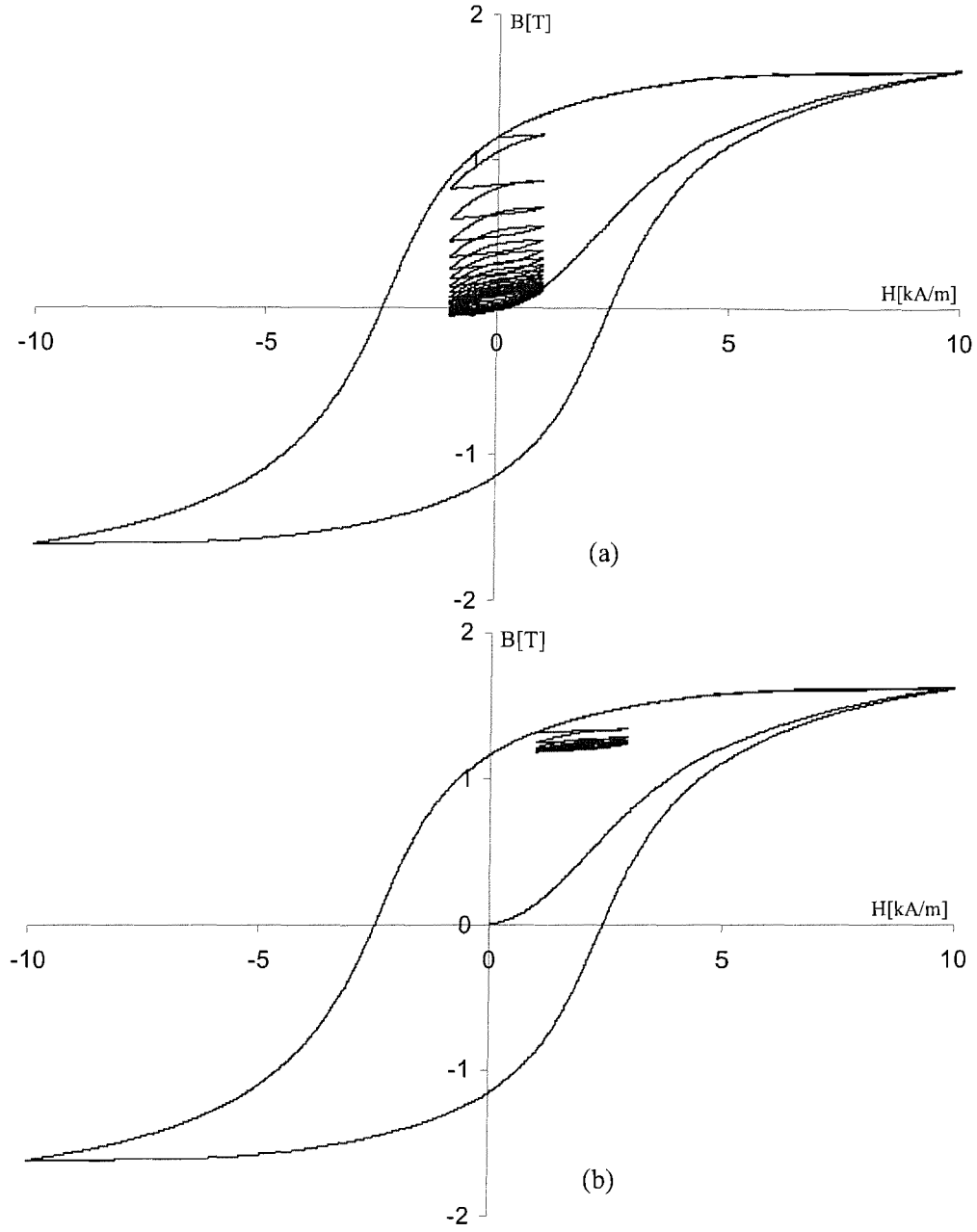

FIGURE 7.8: SystemC simulations showing the minor loop behaviour; main loop amplitude - 10 kA/m, minor loop amplitude - 1 kA/m, bias of H in minor loops a) 0kA/m, b) 2 kA/m.

A comparison is done in implementation and in CPU time between SystemC and VHDL-AMS. Both implementations of the JA model in SystemC and VHDL-AMS produced comparable results. Appendix D.3 shows the VHDL-AMS code. Figure 7.8 and Figure 7.9 show virtually the same behaviour for both simulators.

However, with respect to the CPU time, SystemC proved to be faster than VHDL-AMS (see Table 7.2). The simulations were carried out on a PIII PC with 512MB RAM for 110 samples of H and with the step size $dh = 12$ A/m.

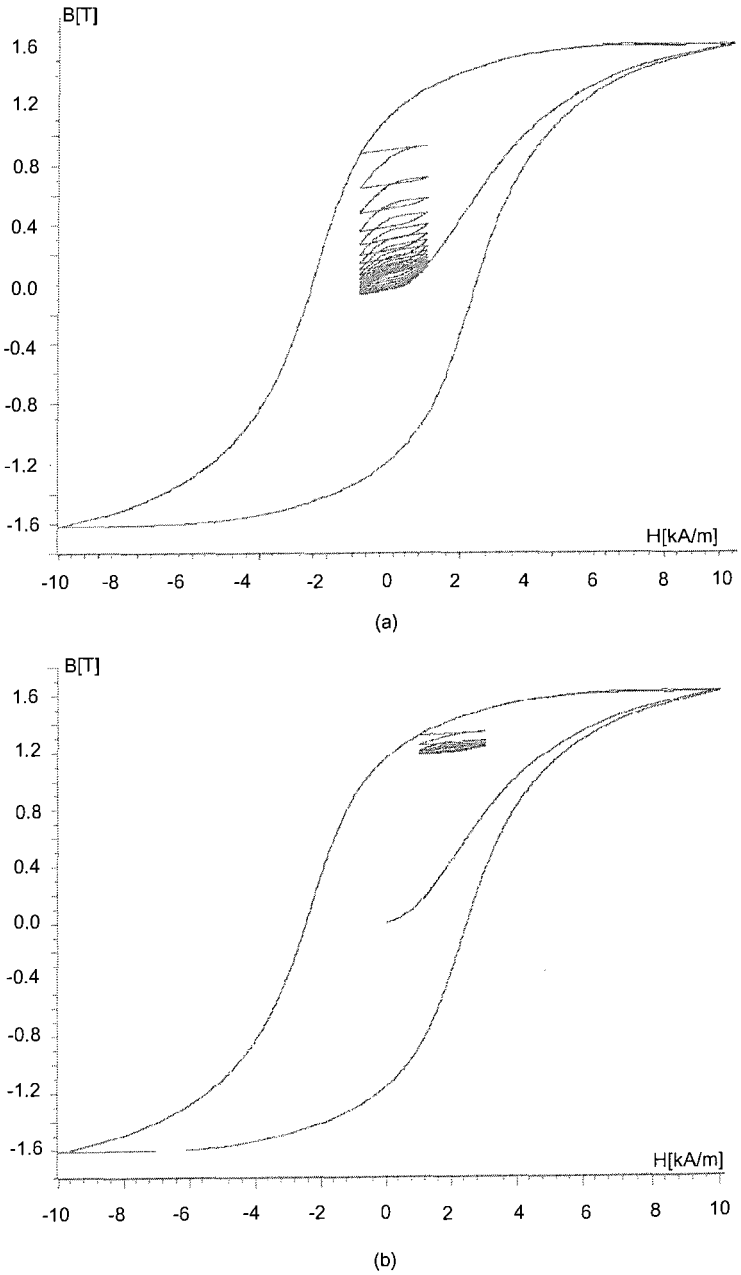

FIGURE 7.9: VHDL-AMS simulations a) BH curve with symmetric minor loops (bias of H is 0kA/m), b) BH curve with asymmetric minor loops, H bias of 2kA/m.

It ought to be reiterated that the Forward Euler and Runge-Kutta methods produced nearly identical results due to a very small sensitivity threshold of dh, which

	Runge-Kutta	Forward Euler
SystemC-A	0.380s	0.370s
VHDL-AMS	31.044s	30.533s

TABLE 7.2: Simulation times of SystemC and VHDL-AMS for ferromagnetic hysteresis.

was about 0.12% of the maximum applied field value. As a consequence, the maximum relative difference in the calculated induction B between the two integration methods was approximately 1%, as shown in Figure 7.10. This suggests that the simple Forward Euler integration method is adequate for this application. The 4^{th} order Runge-Kutta offered virtually no benefit in terms of accuracy, and the related code was more complex and there was a slight CPU time overhead as shown in Table 7.2.

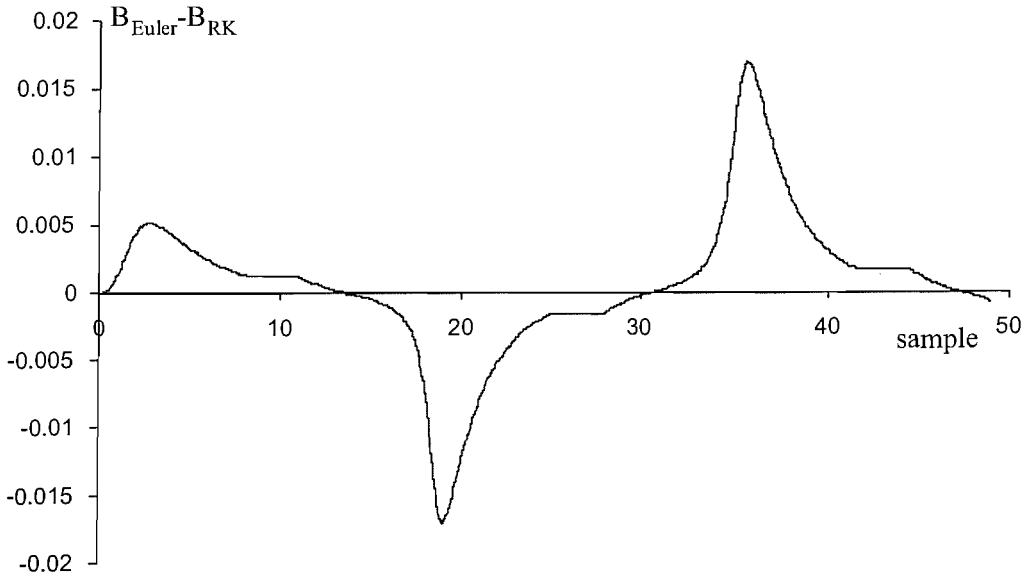

FIGURE 7.10: ΔB the difference between Euler and Runge-Kutta using SystemC.

7.6 Concluding Remarks

This chapter has demonstrated modelling ferromagnetic hysteresis to illustrate SystemC-A capabilities in modelling non-electrical systems. The system is based

on the Jiles-Atherton model and was modelled in SystemC-A using two approaches. The first approach uses the common way of time discretisation of the magnetic slope. This method suffers from numerical difficulty and singularity. Taking the advantage of HDL, the ferromagnetic hysteresis was modelled using a second approach which adopt timeless discretisation. The second approach overcomes some reported problems that have occurred in other implementations, namely long simulation times, non-convergence and numerical instability. The new model was capable of producing minor loops with no numerical difficulties for various minor loops sizes and in different positions. A numerically reliable alternative function to eliminate the well known singularity of Langevin's function was also implemented. The simulation has shown that, due to extra limitations on the step size, both Euler and Runga-Kutta methods produce nearly identical results in terms of accuracy. Hence, the use of Forward Euler is advisable due to its simplicity. For comparison purposes both approaches were implemented in VHDL-AMS showing matching results with SystemC-A models. However, SystemC-A has the advantage of simulation high speed.

Chapter 8

Mixed-domain System Modelling Case Study

In this chapter, SystemC-A is validated by modelling and simulating a mixed-domain case study. The system is automotive seating with vibration isolation. It is nonlinear complicated mixed electrical-mechanical-hydraulic domains. The traditional way of modelling such a combination at component level is to model each domain separately in different languages and/or environments. The system is an excellent choice to validate SystemC-A because it is involving complex DAEs, complex control systems and it is the state of the art in automotive suspension systems. The system to be modelled and simulated is designed originally by Liu and Wagner [139, 140]. They simulated the system in non-HDL environment (Matlab and Simulink) [13]. The system was also modelled in VHDL-AMS by Wang and Kazmierski [141].

The use of HDL in automotive design has been started recently and SystemC-A would be an excellent environment to model mixed-domain systems. It is because SystemC-A models systems in hierarchal analogue or digital modules which represent different parts of the system connected together through ports and signals.

The remainder of this chapter is organised as follows: Section 8.1 describes briefly the vibration isolation seating system and illustrating the mathematical representations of its main parts, the chassis and seating, the actuator and the controllers. A suite of three different types of controllers are used to regulate the automotive seat when subjected to road disturbances. The controllers are Proportional-Integral Controller (PIC), Variable Structure Controller (VSC) and Optimal Controller (OC). Section 8.2 illustrates the implementation of the system in SystemC-A for different stimuli. The results are discussed and compared with VHDL-AMS simulations. Finally, Section 8.3 gives the conclusion.

8.1 Vibration Isolation Seating System

The attenuation of road disturbances of vehicle occupants is a very important issue in riding quality of light-duty and off-road vehicles. The passengers are subjected to high and prolonged disturbances on rough roads. One strategy is the use of vibration isolation systems to attenuate the vibrations between the passenger seat and the vehicle's floor by placing a well-controlled actuator in between. A good model of the whole system is therefore required to reflect all the details of the actual system. The system consists of three main parts, plant (i.e. the passenger seat and vehicle chassis), electromechanical actuator, and controller as shown in Figure 8.1. There are two sensors which monitor the seat and chassis and hence generate input signals to the controller. The control signal is connected to the actuator which is a force generator introduced to improve ride quality.

8.1.1 Mathematical Model of Chassis and Seating System

As shown in Figure 8.1 the vehicle mass M_c and passenger/seat mass M_s are separated by a passive spring K_s and damper C_s. The seat is further isolated

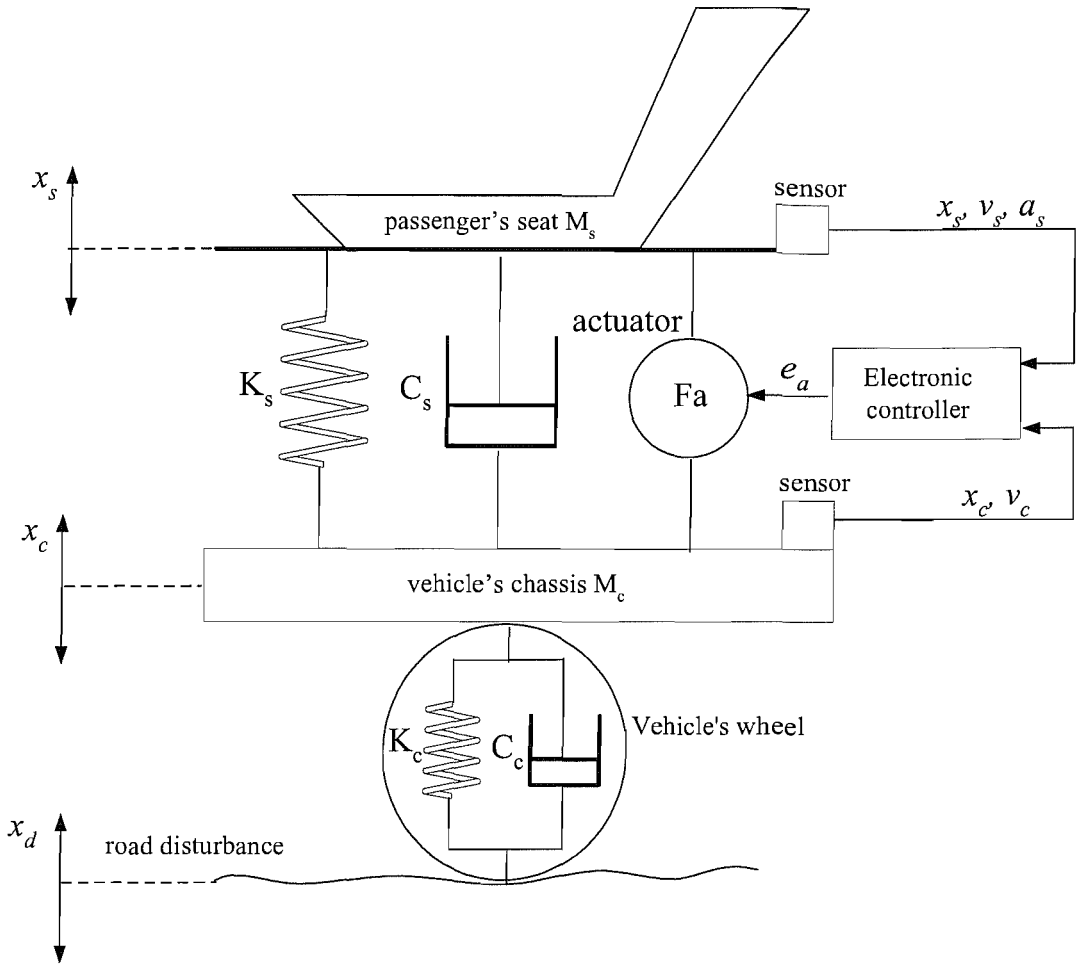

FIGURE 8.1: Vibration isolation seating system.

from the chassis by the force actuator in parallel with the spring and damper. An external displacement x_d represents the system input and acts through a passive spring K_c and damper C_c. The other input is the actuator's force $F_a = A_p \Delta P$. The equation of motion for the seat can be written as:

$$\frac{d^2x_s}{dt^2} = -\frac{C_s}{M_s}\left(\frac{dx_s}{dt} - \frac{dx_c}{dt}\right) - \frac{K_s}{M_s}(x_s - x_c) - \frac{A_p}{M_s}(P_1 - P_2) \tag{8.1}$$

where x_s and x_c are the seat and chassis displacement respectively and P_1 and P_2 are the pressures in upper and lower actuator chambers respectively. Similarly,

the equation of motion for the chassis can be expressed as:

$$\frac{d^2x_c}{dt^2} = -\frac{C_s}{M_c}(\frac{dx_c}{dt} - \frac{dx_s}{dt}) - \frac{K_s}{M_c}(x_c - x_s) - \frac{K_c}{M_c}(x_c - x_d) - \frac{C_c}{M_c}(\frac{dx_c}{dt} - \frac{dx_d}{dt}) + \frac{A_p}{M_c}(P_1 - P_2) \quad (8.2)$$

The output variables are selected to be the relative velocity V_{rel} and the relative displacement x_{rel} which defined in Eq.8.3 and 8.4.

$$V_{rel} = \frac{dx_s}{dt} - \frac{dx_c}{dt} \quad (8.3)$$

$$x_{rel} = x_s - x_c \quad (8.4)$$

Table 8.1 lists and defines the seat and chassis parameters together with their values which are used in the simulations.

Symbol	Definition	Value
M_c	mass of vehicle chassis (kg)	$1.46e+03$
M_s	mass of passenger seat (kg)	$1.0e+02$
K_c	chassis spring stiffness (N/m)	$7.492e+04$
K_s	seat spring stiffness (N/m)	$3.002e+04$
C_c	chassis damping (N.sec/m)	$5.82e+03$
C_s	seat damping (N.sec/m)	$1.1e+03$
A_p	effective piston face area (m^2)	$2.115e-03$

TABLE 8.1: Chassis and seat model parameters.

8.1.2 Mathematical Model of Actuator

The actuator is an electromechanical hydraulic system, which operates in parallel with springs and dampers. It consists of a DC motor, some mechanical parts (such as gear train and rack) and a hydraulic vibration absorber as illustrated in Figure 8.2 and Figure 8.3. The actuator input from the controller is a DC voltage (e_a), which drives the motor to output a rotational torque (T_m). The gear train transmits the rotational velocity from the DC motor to the rack.

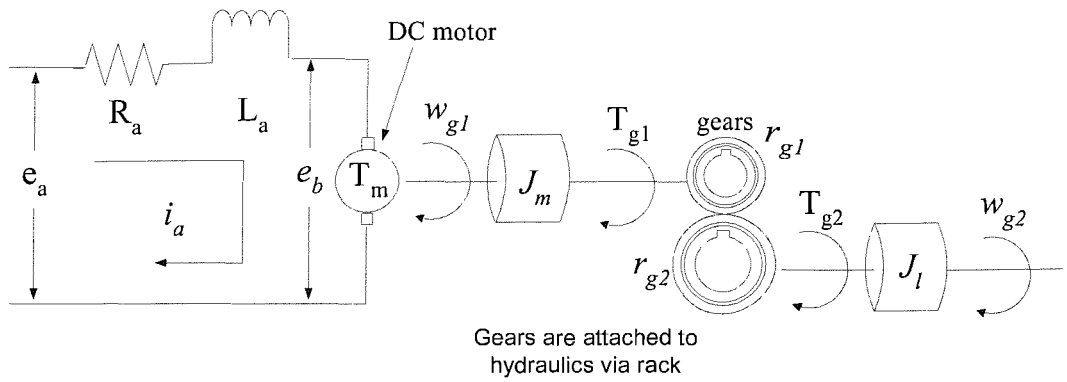

FIGURE 8.2: Actuator's DC motor and gear mechanisation.

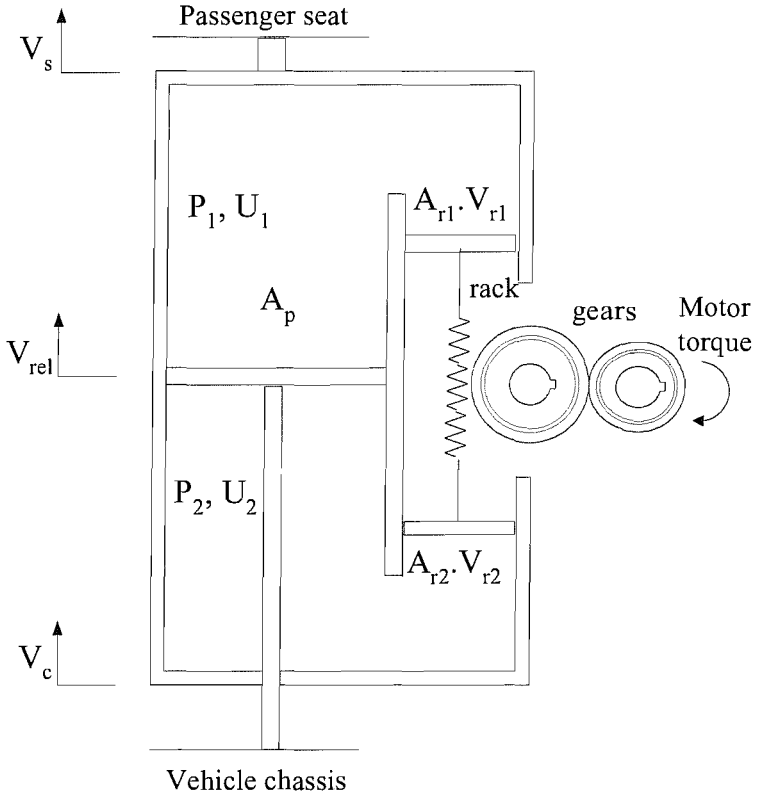

FIGURE 8.3: Actuator's hydraulic mechanisation.

The rack converts the motors's rotational motion into translational velocity, which impacts the pressures of the upper and lower chambers of the hydraulic piston (P_1 and P_2 respectively). The vehicle's chassis is attached to the hydraulic cylinder's piston rod and the seat sits on the cylinder cap. The actuator force is dependent on the pressure difference between the upper and lower chambers ΔP. The generated force attenuates the vibrations by acting on the vehicle's chassis and passenger's

seat.

The actuator's DAEs are from electrical, mechanical and hydraulic domains. The DC motor develops a torque T_m which is proportional to the armature current i_a, where K_t is motor-torque constant.

$$T_m = K_t i_a \tag{8.5}$$

When the armature rotates, a back emf (e_b) is induced and proportional to the flux and angular velocity $\frac{d\theta_{g1}}{dt}$ as defined in Eq.8.6, where K_b is a constant.

$$e_b = K_b \frac{d\theta_{g1}}{dt} \tag{8.6}$$

Applying Kirchoff's voltage law to the electric circuit of the armature,

$$L_a \frac{di_a}{dt} + R_a i_a + e_b = e_a \tag{8.7}$$

Applying Newton's law to the input rotational system dynamics (J_m),

$$J_m \frac{d^2\theta_{g1}}{dt^2} + b_m \frac{d\theta_{g1}}{dt} + T_{g1} = T_m \tag{8.8}$$

Ideal gears are assumed by neglecting friction losses and gear mass, hence the equations of input and output angular velocities ($\omega_{g_1}, \omega_{g_2}$) and torques ($T_{g1}, T_{g2}$) can be written as in Eq.8.9 and Eq.8.10.

$$\frac{\omega_{g2}}{\omega_{g_1}} = \frac{r_{g1}}{r_{g2}} \tag{8.9}$$

$$\frac{T_{g1}}{T_{g2}} = \frac{\omega_{g2}}{\omega_{g1}} \tag{8.10}$$

Applying Newton's law to the load shaft (J_l) gives Eq.8.11.

$$J_l \frac{d^2\theta_{g2}}{dt^2} + b_l \frac{d\theta_{g2}}{dt} + T_L = T_{g2} \tag{8.11}$$

The rack's linear velocity (V_{r2}) can be determined as in Eq.8.12, and $V_{r2} = V_{r1}$ since the two hydraulic pistons are connected.

$$V_{r2} = \omega_{g2} r_{g2} \tag{8.12}$$

The torque (T_L) on the load shaft (J_l) is determined in Eq.8.13.

$$T_L = (A_{r2}P_2 - A_{r1}P_1) r_{g2} \tag{8.13}$$

The hydraulic pressure in the upper and lower actuator chamber (shown in Figure 8.3) are described in Eq.8.14 and Eq.8.15.

$$\frac{dP_1}{dt} = A_p V_{rel} - A_{r1} V_{r1} (\frac{\beta_1}{U_1}) \tag{8.14}$$

$$\frac{dP_2}{dt} = -A_p V_{rel} + A_{r2} V_{r2} (\frac{\beta_2}{U_2}) \tag{8.15}$$

Table 8.2 lists and defines the actuator parameters together with their values which are used in the simulations.

8.1.3 Controllers

In order to attenuate road vibrations, a suite of different types of controllers were designed including linear, nonlinear, and intelligent designs. The controllers are Proportional-Integral (PIC), Variable Structure (VSC) and Optimal Controller (OC). Inputs to the controllers are the dynamic seat and chassis motions i.e. the displacement, velocity and acceleration of the passenger seat (x_s, v_s, a_s), the displacement and velocity of the vehicle chassis (x_c, v_c). Any single controller may

Symbol	Definition	Value
K_t	motor torque constant (N.m/amp)	$9.15e-02$
K_b	back emf constant (V.sec/rad)	$5.20e-02$
L_a	armature inductance (H)	0.0
R_a	armature resistance (Ω)	1.0
N	Gear ratio	1.0
J_m	moment of inertia of motor (Kg.m^2)	$2.119e-06$
β_1	fluid bulk modulus of upper chamber (N/m^2)	$8.61e+07$
b_m	viscous friction of motor (N.m/(rad/sec))	$7.06e-04$
r_{g1}	radius of gear 1 (m)	$2.54e-02$
r_{g2}	radius of gear 2 (m)	$2.54e-02$
J_l	moment of inertia of load shaft (Kg.m^2)	$2.119e-06$
β_2	fluid bulk modulus of lower chamber (N/m^2)	$8.81e+07$
b_l	viscous friction of load shaft (N.m/(rad/sec))	$7.06e-04$
A_p	effective piston face area (m^2)	$2.115e-03$
A_{r1}	area of upper rack end (m^2)	$5.067e-05$
A_{r2}	area of lower rack end (m^2)	$5.067e-05$

TABLE 8.2: Actuator model parameters.

have any set of inputs. Output of the controller is the voltage sent to the DC motor (e_a). Reference [140] describes in details the controllers designs.

The general state space model of the system to be controlled is given in Eq.8.16. It is required for designing the controllers.

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{8.16}$$

The plant's dynamics are represented in the state-space form as in Eq.8.17 and Eq.8.18, where $x_s, x_c, \dot{x_s}, \dot{x_c}$ are the plant's states.

$$\begin{bmatrix} \dot{x_s} \\ \dot{x_c} \\ \ddot{x_s} \\ \ddot{x_c} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{K_s}{M_s} & \frac{K_s}{M_s} & -\frac{C_s}{M_s} & \frac{C_s}{M_s} \\ \frac{K_s}{M_c} & -\frac{(K_s+K_c)}{M_c} & \frac{C_s}{M_c} & \frac{(C_s+C_c)}{M_c} \end{bmatrix} \cdot \begin{bmatrix} x_s \\ x_c \\ \dot{x_s} \\ \dot{x_c} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{K_c}{M_c} & \frac{C_c}{M_c} \end{bmatrix} \cdot \begin{bmatrix} x_d \\ \dot{x_d} \end{bmatrix} \tag{8.17}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_s \\ x_c \\ \dot{x_s} \\ \dot{x_c} \end{bmatrix} \tag{8.18}$$

8.1.3.1 Proportional-Integral Controller PIC

The PIC is used to act on the error between the seat's set-point acceleration and the actual value ($e = a_{sp} - a_s$). Through the integral operator, the PIC brings quickly the error signal to zero. Eq.8.19 defines the PIC equation, where $K_p = 13.5Vs^2/m$ and $K_I = 0.27Vs/m$ [140] are the proportional and integral gains respectively. They are selected using analytic and trial/error processes.

$$e_a = K_p\, e + K_I \int e\, dt \tag{8.19}$$

8.1.3.2 Variable Structure Controller VSC

The VSC relies on a high-speed switching feedback strategy to establish a robust control for uncertain plant models. The switching control algorithm drives the plant's state trajectory to a user selected switching line and then maintains the trajectory at that line. The switching line is chosen such that the system motion exhibits the desired stability and/or tracking characteristics. A switching line $\sigma(x)$ is defined as in Eq.8.20,

$$\sigma(x) = Cx = C_1x_1 + C_2x_2 + ... + C_nx_n = 0 \tag{8.20}$$

where x is the plant's states vector and C is a vector of constants which determine the slope of the switching line in the phase plane. The line $\sigma = 0$ describes the system average behaviour with chosen dynamics. The first step in the controller design is to choose a Lyapunov's function (V) as in Eq.8.21 to guarantee stability of the switching line.

$$V = \sigma^2 \tag{8.21}$$

where σ^2 is positive and nonzero everywhere except on the switching line ($\sigma = 0$). The switching line should satisfy the following condition:

$$\dot{V} = 2\sigma\dot{\sigma} < 0 \tag{8.22}$$

which implies that,

$$\sigma\dot{\sigma} < 0 \tag{8.23}$$

solving $\dot{\sigma} = C_1\dot{x}_1 + C_2\dot{x}_2 + C_3\dot{x}_3 + C_4\dot{x}_4$, gives $C = [1.72, 262.4, 4.5, 65.7]$ [140].

After substituting and reordering Eq.8.23, the inequality relations for the control signal is obtained as follows:

$$u = \begin{cases} u - \delta \quad for \quad \sigma > 0 \\ u + \delta \quad for \quad \sigma < 0 \end{cases} \tag{8.24}$$

The second step is to compute the controller's feedback gains which derive the plant's trajectories to the sliding surface. The full state feedback has the form of $u = Kx = k_1 x_1 + ...k_n x_n$, with individual gains defined in Eq.8.25

$$k_i = \begin{cases} < k_{i1}, \quad if \quad \sigma(x).x_i > 0 \\ > k_{i2}, \quad if \quad \sigma(x).x_i < 0 \end{cases} \tag{8.25}$$

where k_{i1} and k_{i2} represent the maximum and minimum limits on each gain. The gains are selected as k_1=-94.8 N/m, k_2=-5700 N/m, k_3=-1440 Ns/m, and k_4=437 Ns/m.

8.1.3.3 Optimal Controller OC

The optimal controller is based on full state feedback Linear Quadratic Regulator (LQR). LQR controller problem is to find a control law $u = e_a = -Kx$, such that $(A - BK)$ is stable and minimises a specified linear quadratic performance index defined in Eq.8.26. In Eq.8.26, Q and R are the state and input weighting matrices, respectively. The performance index is selected based on a balanced tradeoffs between convergence speed to the system states and the input amplitudes.

$$J(x, u, Q, R) = \frac{1}{2} \int_0^\infty (x^T Q \; x + u^T R \; u) dt, \quad Q \geq 0, \; R > 0 \tag{8.26}$$

The vector of feedback gains K is to be calculated using Eq.8.27.

$$K = R^{-1}B^T P \tag{8.27}$$

where P can be found by solving Riccati equation defined in Eq.8.28.

$$PA + A^T P - PBR^{-1}B^T P + Q = 0 \tag{8.28}$$

The controller gains are found to be K=[-116 -6520 -1670 -460] by using MATLAB's function ([K,P,E]=lqr(A,B,Q,R)), where E is the eigenvalues matrix of the closed loop system with optimal state feedback which determine the stability of the system.

8.2 Modelling and Simulation

The automotive system was modelled in SystemC-A making use of the modular modelling where the main parts of the system (plant, actuator and controller) were modelled as SystemC-A components at behavioural level as illustrated in Figure 8.4. The testbench (SystemC *SC_MODULE*) includes instances of all parts connected together by signals. The input stimulus is generated by a sinusoidal voltage source and connected to the chassis and seating module via nodes as shown in Listing 8.1.

```
1  void testbench::system(){
2      // global connecting signals
3      sc_signal<double> Vrel,as,deltaP,ea;
4      // instantiating nodes and components
5      n2 = new Node("0");
6      n1 = new Node("n1");
7      sc_a_voltageS_sin *I1= new sc_a_voltageS_sin("Vsin",n1,n2,5.125,0,1,0,0);
8      seating      *s1     = new seating("s1",n1,&deltaP,&Vrel,&as);
9      actuator     *act1   = new actuator("act1",&Vrel,&ea,&deltaP);
10     PI           *pi1    = new PI("pi1",&as,&ea);
11
12     sc_start(2.5,SC_SEC); \\start simulation for 2.5Sec
13 }
```

LISTING 8.1: SystemC-A testbench of the automotive vibration isolation system.

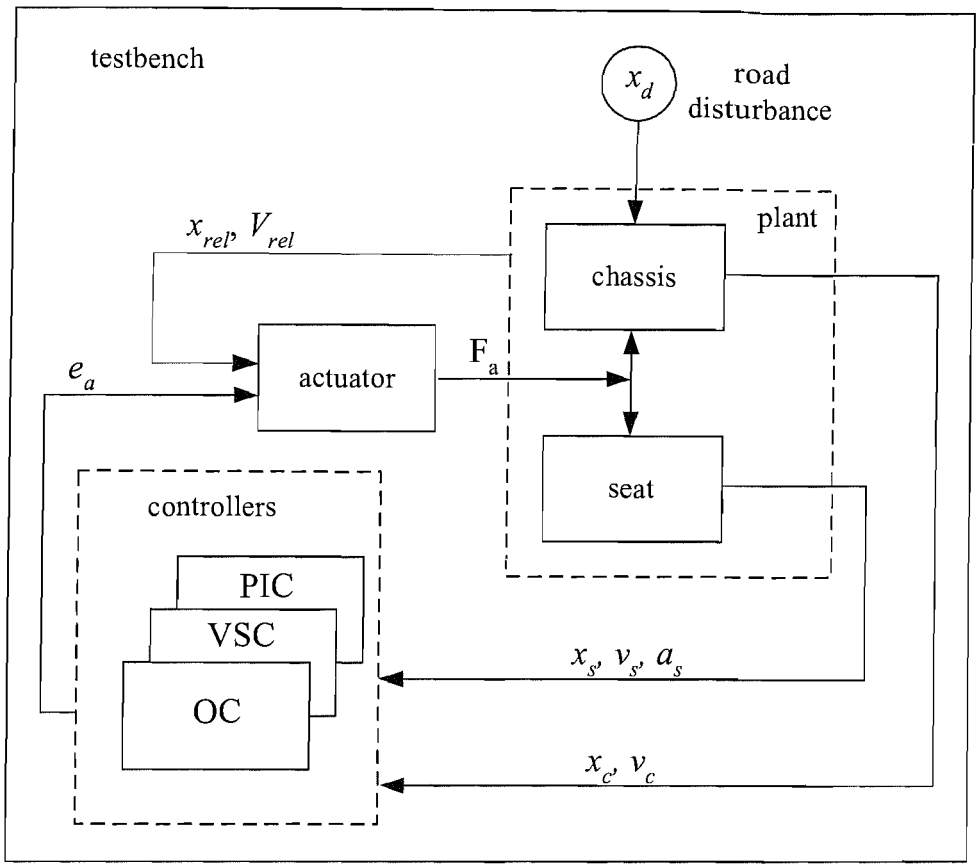

FIGURE 8.4: Block diagram of the automotive system implementation in SystemC-A.

Listing 8.2 gives SystemC-A model of the chassis and passenger's seat, whereas Listing 8.3 gives the actuator model.

```
1   ...
2   seating::seating(char nameC[5], sc_a_system_variable *node_a,
3       sc_signal<double>*deltaP1,sc_signal<double>*Vrel1,
4       sc_signal<double>*as1):sc_a_component(nameC,node_a, 0,0){
5           deltaP_sig=deltaP1;
6           Vrel_sig=Vrel1;
7           as_sig=as1;
8       xcQ = new sc_a_free_variable("xcQ"); //system variables
9       xsQ = new sc_a_free_variable("xsQ");
10      ysQ = new sc_a_free_variable("ysQ");
11      ycQ = new sc_a_free_variable("ycQ");
12  }
13
14  void seating::BuildB(){
15      ...
16      xd=X(a);//first input: road disturbance
17      deltaP=deltaP_sig->read();//second input: deltaP
18      Vrel=ys-yc;
19      Vrel_sig->write(Vrel); //first output: relative velocity
20                             // between chassis and seat
21      as=Xdot(ysQ);
22      as_sig->write(as);//second output: acceleration of seat
23      // the four equations of chassis and seat
24      Equation(ysQ, -dysdt - (Cs/Ms)*(ys-yc) - (Ks/Ms)*(xs-xc) - (Ap/Ms) * deltaP);
25      Equation(xsQ, -dycdt - (Cs/Mc)*(yc-ys) - (Ks/Mc)*(xc-xs) - (Kc/Mc)*(xc-xd) -
```

```
26              (Cc/Mc)*(yc-dxddt) + (Ap/Mc) * deltaP );
27      Equation(xcQ, -dxsdt + ys );
28      Equation(ycQ, -dxcdt + yc );
29  }
```

LISTING 8.2: SystemC-A implementation of the chassis and seating.

```
1  actuator::actuator(char nameC[5],sc_signal<double>*Vrel1,
2              sc_signal<double> *ea1,sc_signal<double>*deltaP1):
3      sc_a_component(nameC,node_a, 0,0){
4      Vrel_sig=Vrel1;
5      ea_sig=ea1;
6      deltaP_sig=deltaP1;
7      //system variables
8      TmQ      = new sc_a_free_variable("TmQ");
9      ebQ      = new sc_a_free_variable("ebQ");
10     TLQ      = new sc_a_free_variable("TLQ");
11     VrQ      = new sc_a_free_variable("VrQ");
12     iaQ      = new sc_a_free_variable("iaQ");
13     wg1Q     = new sc_a_free_variable("wg1Q");
14     wg2Q     = new sc_a_free_variable("wg2Q");
15     U1Q      = new sc_a_free_variable("U1Q");
16     U2Q      = new sc_a_free_variable("U2Q");
17     P1Q      = new sc_a_free_variable("P1Q");
18     P2Q      = new sc_a_free_variable("P2Q");
19     TgQ      = new sc_a_free_variable("TgQ");
20     deltaPQ = new sc_a_free_variable("deltaPQ");
21 }
22
23 void actuator::IC(void){// initial values
24     InitialC(U1Q,1.0e-4);
25     InitialC(U2Q,1.0e-4);
26     InitialC(P1Q,1.0e10);
27     InitialC(P2Q,1.0e10);
28 }
29
30 void actuator::BuildB(){
31     ...
32     Vrel=Vrel_sig->read();// first input: relative velocity
33     ea=ea_sig->read();  // second input: acceleration of seat
34     deltaP_sig->write(deltaP);// output: deltaP
35     // DAEs
36     Equation(TmQ, -Tm+Kt*ia);
37     Equation(ebQ, -eb+Kb*wg1);
38     Equation(TLQ, -TL + (Ar2*P2 - Ar1*P1)*rg2);
39     Equation(VrQ, -Vr + wg2*rg2);
40     Equation(iaQ, -ea + eb + Ra*ia + La*diadt);
41     Equation(TgQ, -Jm*dwg1dt- bm*wg1-Tg+Tm);
42     Equation(wg1Q, wg2/N - wg1);
43     Equation(wg2Q, -J1*dwg2dt- b1*wg2-TL+Tg);
44     Equation(U1Q, -dU1dt + Ap*Vrel - Ar1*Vr);
45     Equation(U2Q, -dU2dt -Ap*Vrel + Ar2*Vr);
46     Equation(deltaPQ, -deltaP+(P1-P2));
47     Equation(P1Q, -dP1dt + beta1*(Ap*Vrel - Ar1*Vr)/U1);
48     Equation(P2Q, -dP2dt + beta2*(-Ap*Vrel + Ar2*Vr)/U2);
49 }
```

LISTING 8.3: SystemC-A implementation of the actuator.

In Listings 8.2 and 8.3, the component's constructor defines the number and type of the component's inputs and outputs for each controller. Also in the constructor, system variables are defined as well as the system constants. *BuildB()* functions include DAEs of the system which hint that the system modeled at behavioural

level. The chassis and seating module contains 4 system variables (lines 8-11 of Listing 8.2) which need 4 *Equation()* functions (lines 24-28) to be defined in *BuildB()*, while the actuator has 13 system variables and needs 13 *Equation()* functions to be defined.

To demonstrate one of the controllers representation in SystemC-A, Listing 8.4 gives part of the implementation of the VSC. The model has no system variables and it just involves sequential statements. The inputs to the VSC are the plant's four states ($x_s, x_c, \dot{x}_s, \dot{x}_c$) while the output is the voltage e_a sent to the DC motor.

```
1  void VS::BuildB(){
2   // controller parameters:
3    K11= -95.8;    //K11  K12 are lower and upper limits of K1
4    K12= -93.8;
5    K21= -5704.6; //K21  K22 are lower and upper limits of K2
6    K22= -5702.6;
7    K31= -1446.2; //K31  K32 are lower and upper limits of K3
8    K32= -1444.2;
9    K41= 436.3;    //K41  K42 are lower and upper limits of K4
10   K42= 438.3;
11   c1= 1.72;
12   c2= 262.4;
13   c3= 4.5;
14   c4= 65.7;
15   k_head= 6.648e-02;
16
17   sigma = c1*xc + c2*xs + c3*xcdot + c4*xsdot;
18   p1 = sigma*xc;
19   p2 = sigma*xs;
20   p3 = sigma*xcdot;
21   p4 = sigma*xsdot;
22     if (p1>0.0)
23         K1 = K11;
24     else
25         K1 = K12;
26
27     if (p2 > 0.0)
28         K2 = K21;
29     else
30         K2 = K22;
31
32     if (p3 > 0.0)
33         K3 = K31;
34     else
35         K3 = K32;
36
37     if (p4 >0.0)
38         K4 = K41;
39     else
40         K4 = K42;
41
42     ea = k_head*(xc*K1 + xs*K2 + xcdot*K3 + xsdot*K4);
43     ea_sig->write(ea);//controller output voltage sent to the DC motor
44     }
```

LISTING 8.4: SystemC-A implementation of the variable structure controller.

The analogue simulator used a time step of 0.1m seconds. The system was simulated using two types of input stimuli: a single jolt sine wave and multiple sine waves with a White Gaussian Noise (WGN) (standard deviation =1cm). The two stimuli represent road disturbances x_d at frequency of 5.162Hz and amplitude of x_{dp-p}=10cm. The most important variable to be monitored is the maximum value of passenger's seat displacement x_{sp-p}. The passive system is first simulated to study the effect of applying different types of controllers to the system.

8.2.1 Single Jolt Simulation

Simulations of a single jolt sin wave were carried out for a duration of 2.5 seconds. The seat displacement response of the passive system together with responses when using different controllers are shown in Figure 8.5. The passive spring and damper can attenuate road disturbances x_{dp-p} from x_{sp-p} = 10cm to x_{sp-p} = 3.05cm. Compared to the passive system response, the optimal controller achieved the best isolation performance with a factor of 5.32 reduction in x_{dp-p}. Whereas the PIC and VSC achieved 2.02 and 4.54 respectively.

8.2.2 Multiple Sin Waves with WGN Simulation

Simulations of multiple sin waves with WGN were carried out for 1.0 second. The seat displacement response of the passive system together with responses when using different controllers are shown in Figure 8.6. The passive spring and damper can attenuate road disturbances x_{dp-p} from x_{sp-p} = 10cm to x_{sp-p} = 2.58cm. Compared to the passive system response, the optimal controller again achieved the best isolation performance with 4.26 reduction in x_{dp-p}. Whereas the PIC and VSC achieved 1.92 and 3.47 respectively. Reduction in performance was expected under multiple sin waves and noisy stimulus.

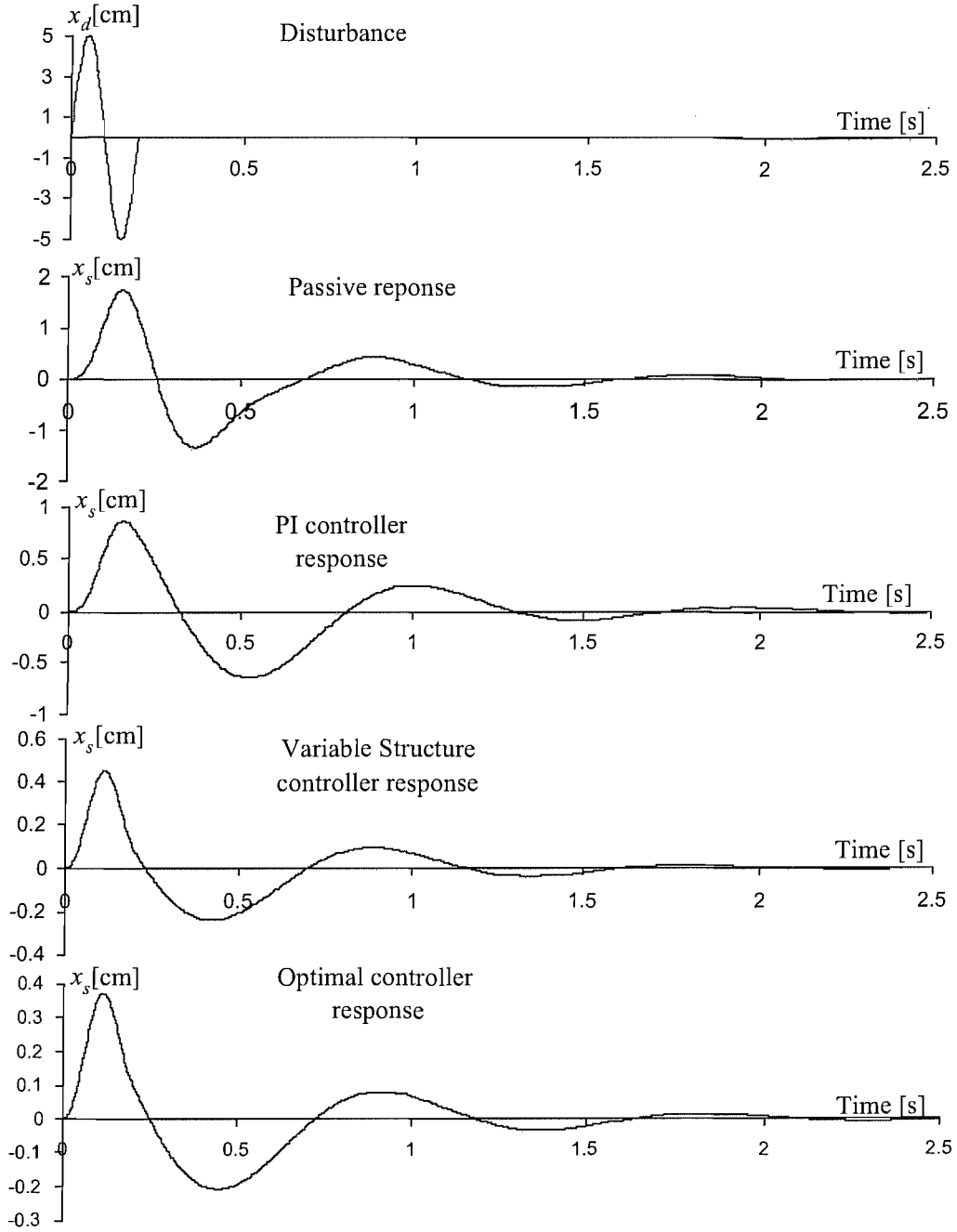

FIGURE 8.5: Single jolt sine wave disturbance simulation with responses of the three controllers.

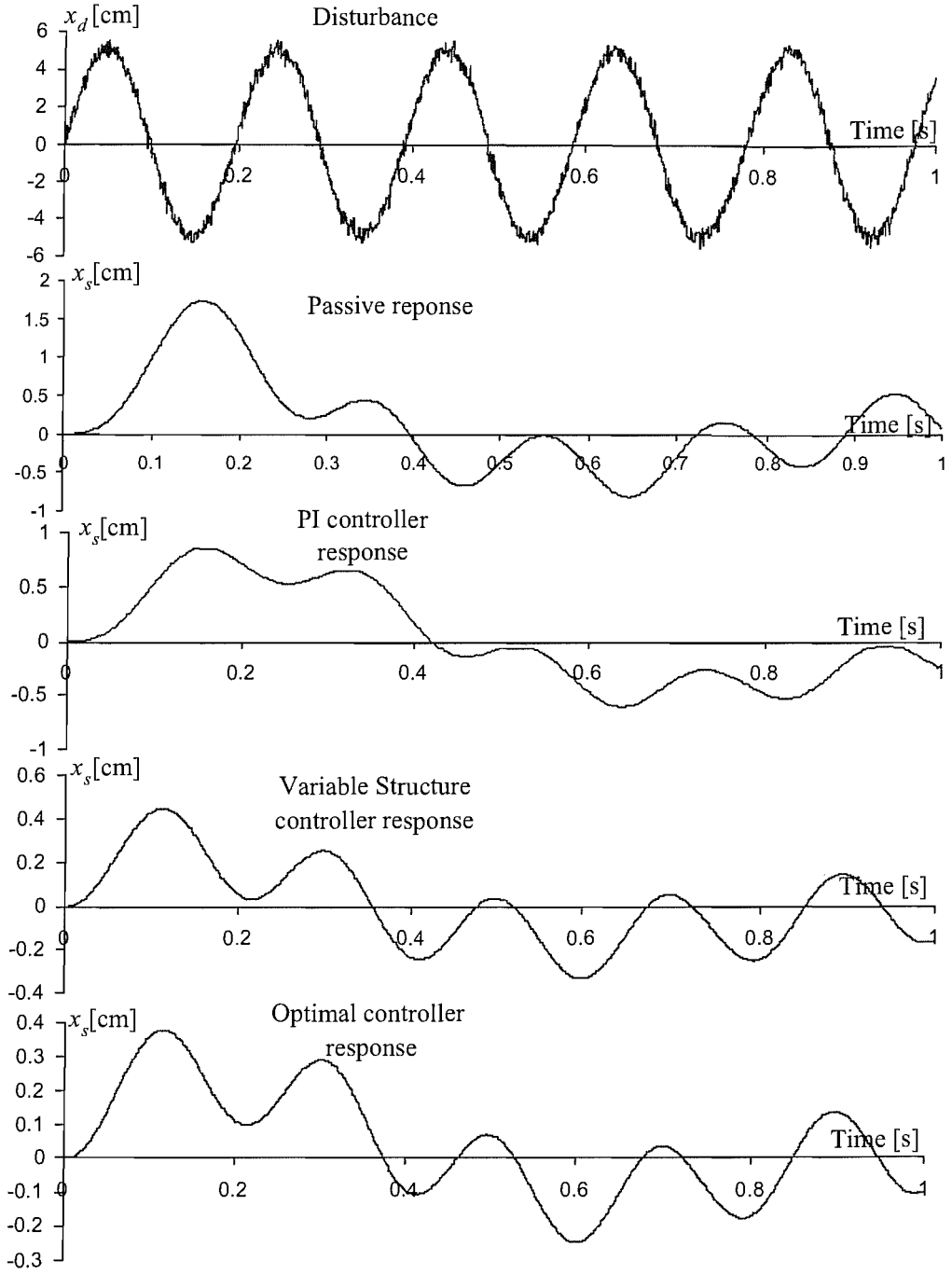

FIGURE 8.6: Noisy sine wave disturbance simulation with responses of the three controllers.

8.2.3 Comparison with VHDL-AMS

The model of the vibration isolation seating system was modelled and simulated in VHDL-AMS [141]. SystemC-A shows accurate results by comparing its simulation performance of x_{sp-p} with VHDL-AMS results in Table 8.3. The maximum relative percentage differences in the maximum value of seat position x_{sp-p} between SystemC-A and VHDL-AMS simulations was approximately 0.860% for the optimal controller under single jolt sin wave, and it was 2.642% for the passive system under multiple sin waves with WGN simulation.

TABLE 8.3: SystemC-A and VHDL-AMS performance figures of the seat position x_{sp-p} (cm) for the passive system and the suite of controllers.

	SystemC-A		**VHDL-AMS**		**% Relative error**	
	single jolt sin wave	sin waves with WGN	single jolt sin wave	sin waves with WGN	single jolt sin wave	sin waves with WGN
Passive system	3.05	2.58	3.07	2.65	0.68%	2.64%
PIC	1.51	1.34	1.52	1.37	0.59%	2.04%
VSC	0.68	0.74	0.69	0.74	0.58%	0.67%
OC	0.57	0.60	0.58	0.60	0.86%	0.17%

8.3 Concluding Remarks

This chapter has demonstrated modelling automotive vibration isolation seating system to illustrate SystemC-A ability in modelling mixed-domain systems. The system is nonlinear complicated mixed electrical-mechanical-hydraulic domains providing a challengeable task for SystemC-A validation. The system was modelled at behavioural level with three types of controllers, proportional-integral, variable structure and optimal controllers. The system was tested with two types of stimuli, a single jolt sine wave and multiple sine waves with a white Gaussian noise. Results were studied and the optimal controller has shown the best reduction in road disturbance for both input stimuli. Further, SystemC-A simulations were compared to published VHDL-AMS simulations, showing highly comparable

numerical figures, which proves that SystemC-A can be compared to well established HDLs.

Chapter 9

Conclusions and Future Research

The main aim of this research stems from the modelling difficulties of analogue and mixed-signal systems which have been facing the design community, as explained in Chapter 1. This research presents a new mixed-signal language capable of simulating a variety of general analogue models suitable for a number of application areas and levels of abstraction with special emphasis on high level designs. The developed language is an extension of SystemC and has been named SystemC-A. The objective was achieved in a number of steps.

In Chapter 2 the state of the art SystemC high level digital language was investigated and studied in order to provide the requirement for an efficient digital simulator as part of the aimed AMS modelling environment. Further, VHDL-AMS language elements were overviewed to be an inspiration for this research AMS extension new elements.

The next step was the development of the AMS extension which represents the first contribution of this thesis. It is described in Chapter 3 and 4. In Chapter 3, new language syntax and constructs were developed to represent *analogue system variables* such as *node* variable to describe general circuit nodes, flow variable to

describe MNA-like flow variables, and *free variables*, to represent variables in user-defined differential and/or algebraic equations. All system variables represent the unknowns in the set of DAEs and can be extended to user-defined type variables. Another developed element is a construct representation of an *analogue component* which can be extended to circuit-level components or user-defined components. A library of circuit-level components was developed which provides fundamental components such as resistor, capacitor, inductor, diode, transistors and several kinds of voltage and current sources. Further more complex components can be derived using C++ inheritance mechanism. System variables and analogue component classes are to be used to build general analogue systems at circuit-level or system-level. Important analogue-digital interfaces are developed for different connection types and associated issues such as small time step sizes and analogue stepping are solved.

Chapter 4 presents an efficient implementation of an analogue kernel to solve the analogue part of the simulated system. An object-oriented equation formulation method, called the Object-Oriented Newton-Quasi Newton method (OO-NQN), has been developed. A new SystemC-A element called *build* method is used to implement the OO-NQN method. The OO-NQN has two approaches to equation formulation associated with constructing the Jacobian. The first approach is to build exact Jacobian's elements using MNA component's stamps and hence, solve the system using the Newton Method. The other approach is mixed where some Jacobian elements are approximated using the secant method and hence solve the system by means of a Quasi-Newton approach. In the case studies, shorter simulation times were obtained where an exact Jacobian was used. However, the advantage of secant approximations is easiness and less expertise needed to create MNA Jacobian stamps, as well as simpler code. Therefore, OO-NQN provides a flexible means to compromise between simple modelling and simulation speed.

In Chapter 5, an efficient version of the lock-step synchronisation technique has

been developed and represents the second contribution of this thesis. The lock-step technique was implemented, tested and found to be the most suitable synchronisation technique to interface the analogue and digital parts modelled in the same language. The SystemC-A lock-step technique handles efficiently zero step-sizes, which occur frequently in a mixed-signal HDL, whose digital kernel may produce repeated delta cycles.

The new language elements and SystemC-A methods developed in Chapter 3, 4 and 5 were tested, verified and optimised. For this purpose, a wide range of examples were carefully chosen from different physical domains in order to validate SystemC-A from different aspects. The modelling and simulations of the case studies in Chapter 6, 7 and 8 is the third contribution of this thesis.

In Chapter 6, a suite of four electrical case studies was modelled and simulated. The first two case studies were Van Der Pol oscillator and Lorenz chaos. They were modelled to illustrate the ability of SystemC-A to handle behavioural-level specifications using its new language constructs. The other case studies were a Switched Mode Power Supply (SMPS) and a 2GHz Phase Locked Loop (PLL)-based frequency multiplier. The two examples are highly complex mixed-signal systems, which have disparate time scales of their transients, and therefore, require excessive CPU times and might put any standard analogue simulator in difficulties.

All the simulations were successful and no modelling or numerical difficulties were encountered. For the PLL example two noise methods were developed to model and simulate the jitter in the VCO output signal. The first noise method provides a more accurate noise behaviour by using a noise source especially develop to add noise to any component's model. The other noise method provides faster simulations by altering the VCO pulse width directly to form a jitter. This perturbation represents the total effect of any noise source in the PLL. Furthermore, the developed PLL model with the two noise methods has been used to compare

the behaviour and speed of SystemC-A with that of SystemVision VHDL-AMS simulator from Mentor Graphics. There was a simulation speed factor of three in favour of SystemC-A. The main reason behind the speed factor is the adoption of the efficient OO-NQN equation formulation method in SystemC-A, while this option is not provided in the Mentor Graphics simulator.

In Chapter 7, a non-electrical case study of ferromagnetic hysteresis is modelled based on the Jiles-Atherton mathematical model. This system provides real challenges to SystemC-A. The common implementation of the JA model involves numerical integration of a discontinuous and non-linear differential equation. In addition, the model in its original form can sometimes produce a hysteresis curve with negative slopes for which there is no physical justification. Further, the model suffers from convergence problems and long analysis times.

The ferromagnetic hysteresis model was implemented in SystemC-A in two different ways. Firstly, for the purpose of demonstrating, the commonly used model which involves the simulator's analogue solver and suffers from numerical difficulties was implemented and tested. These tests confirmed the usual reported problems as outlined in Section 7.3. Secondly, in the other approach, a new model was developed to overcome most of the encountered numerical problems. A novel, timeless discretisation technique was developed to integrate the magnetisation slope $\frac{dM}{dH}$ without involving the analogue solver. The new model was capable of producing minor loops with no numerical difficulties for various minor loops sizes and in different positions. A numerically reliable alternative function to eliminate the well known singularity of Langevin's function was also implemented. Both approaches were also implemented in VHDL-AMS for verification and comparison. The advantage of SystemC-A over VHDL-AMS in terms of simulation speed was also confirmed by those simulations.

Chapter 8 illustrated the capability of SystemC-A to model complex mixed-physical-domain systems. A mixed-domain automotive case study was modelled and simulated. The system handles dynamics of a passenger seat with vibration isolation and active electronic control. The system is nonlinear and complex using equations from mixed, electrical, mechanical and hydraulic domains. It also involves complex digital controllers to regulate the automotive seat when subjected to road disturbances. The controllers tested were: Proportional-Integral Controller (PIC), Variable Structure Controller (VSC) and Optimal Controller (OC). The system was modelled at the behavioural level using mainly the new equation syntax and OO-NQN equation formulation. Simulations used two types of stimuli: a single jolt sine wave and multiple sine waves with a white Gaussian noise. The optimal controller has shown the best reduction of the road disturbance for both input stimuli. Further, SystemC-A simulations were compared to VHDL-AMS simulations published in the literature, showing highly comparable numerical figures.

It is important to indicate that the developed methods and constructs in this thesis are not restricted to the SystemC language. The work is potentially beneficial to development of analogue kernels for modern AMS HDLs such as VHDL-AMS, Verilog-AMS and other system design languages, including those based on C/C++.

SystemC-A is the first attempt to extend SystemC to provide mixed-signal and mixed-domain modelling capability. A future adoption of SystemC-A may provide significant advantages in the modelling of modern heterogenous Systems on Chip. SystemC-A is an environment to model and simulate systems consisting of digital and analogue parts, as well as hardware/software parts at different abstraction levels, from circuit to concept level. SystemC-A has proved to be a powerful and an easy to learn alternative to existing HDLs. A number of other advantages are listed below:

- Potential for automation: SystemC-A enhances the current design flow by

providing scope for an automated design flow for the analogue blocks and replacing the manual design method.

- High abstraction levels: SystemC-A provides the required high level of design abstraction required for the multi-million transistor era. In addition it also supports low abstraction levels required when modelling critical parts of any system.

- Speed: SystemC-A can simulate complex systems at much higher simulation speeds than those offered by existing HDLs. In PLL example, the CPU time of the model's simulation with SystemC-A is faster by three times than that of VHDL-AMS.

- Powerful modelling features: SystemC-A is based on the C/C++ language which is familiar to most hardware/software designers. Also it has all the properties of general programming languages too. This gives a freedom in modelling and allows for description of very complex AMS systems in a user friendly manner. In many respects, SystemC-A resembles the semantics of standard HDLs. However, it has features which do not have their counterparts in existing HDLs, for example transient noise analysis.

- Model reuse: In SystemC-A, a model can be derived as a derived class of a base model which gives an efficient way of model code reuse and helps significantly in the modelling process.

SystemC-A has many areas which may be subject to further development and extensions. More work is required towards an integrated environment for mixed-signal mixed-domain and multi-discipline applications. Possible further extensions may include alternative descriptions of analogue systems such as frequency domain descriptions, transfer functions or state-space representations. Support for more analogue simulation analyses would be desirable such as AC analysis and Fourier spectral analysis.

In the context of the growing popularity of MEMS applications as well aerospace and automotive virtual prototyping, while SystemC-A already provides support for mixed-physical-domain modelling, it could be further extended to support distributed structures modelled by partial differential equations. Adequate languages for supporting such applications do not yet exist.

Also, the absence of a high level modelling notation for AMS systems is a big hurdle in the development of efficient synthesis methods. This issue needs to be addressed in the near future to aid automation of AMS and mixed-physical-domain design processes. Synthesis methods using a language based on C/C++ could be very flexible and some work has already been done for digital system synthesis [142, 143]. The SystemC-A notation could be a base for AMS and mixed-physical-domain system synthesis. The existing huge C/C++ libraries for numerical optimisation can shorten the development time of such synthesis environments.

Appendix A

Publications

The following are papers published or under review during the course of this thesis work.

1. T. Kazmierski and H. Al-Junaid, "Synchronisation of Analogue and Digital Solvers in Mixed-Signal Simulation on a SystemC Platform," in Proceedings of Forum on Specification and Design Languages FDL, Frankfurt Germany, 23-26 September 2003.

2. H. Al-Junaid and T. Kazmierski, "SEAMS - A SystemC Environment With Analog And Mixed-Signal Extensions," in Proceedings of IEEE International Symposium on Circuit and Systems ISCAS, pages pp. 281-284, Vancouver Canada, 23-26 May 2004.

3. H. Al-Junaid and T. Kazmierski, "An Extension to SystemC to Allow Modelling of Analogue and Mixed-Signal Systems at Different Abstraction Levels," in Proceedings of IEE SoC Design, Test and Technology Seminar, Loughborough University, United Kingdom, 15 September 2004.

4. H. Al-Junaid and T. Kazmierski, "An Analogue and Mixed-Signal Extension to SystemC," IEE Journal of Circuits, Devices and Systems, issue 152, no. 6, pp. 682-690, December 2005.

5. H. Al-Junaid and T. Kazmierski, "HDL Models of Ferromagnetic Core Hysteresis Using Timeless Discretisation of the Magnetic Slope," in Proceedings of Design, Automation and Test in Europe DATE, Munich, Germany, 6-10 March 2006.

6. H. Al-Junaid, T. Kazmierski, Peter Wilson, and Jerzy Baranowski, "Timeless Discretization of the Magnetization Slope in Modeling of Ferromagnetic Hysteresis," IEEE Transaction of Computer Aided Design TCAD, Accepted for publication, 2006.

7. H. Al-Junaid, T. Kazmierski, "System-level hierarchical equation formulation for analogue-mixed signal hardware description languages," under review, IEE letters, 2006.

8. H. Al-Junaid, T. Kazmierski and Leran Wang, "SystemC-A Modelling of an Automotive Seating Vibration Isolation System," under review, Forum on Specification and Design Languages FDL, 2006.

Appendix B

Review of SystemC Applications

In the last few years, SystemC has received much attention from the electronics industry. Researchers have directed their research towards development and improvement of new methodologies including modelling, verification, in addition to synthesis for SoC designs at different abstraction levels. This appendix reviews SystemC applications from the literature in these areas.

B.1 Modelling

There are a large number of success stories of using SystemC in modelling from real users as well as academic researchers. These appeared in most well known conference proceedings such as Design Automation Conference (DAC), Asia and South Pacific Design Automation Conference (ASP-DAC), Design and Verification Conference (DVCON) and Design, Automation and Test in Europe Conference and Exhibition (DATE).

Network SoC from Samsung

Samsung had a network SoC model written in RTL Verilog [144] which was already marketed but it did not meet performance expectations. They remodelled their SoC in SystemC TLM and achieved 99% cycle accuracy compared with the RTL Verilog version and over 100 times faster execution. Most importantly, with SystemC the engineers were able to better explore the system to find and track down its design problems.

Exposure control unit from Bosch and University of Tuebingen

A cooperative project between Robert Bosch GmbH, OFFIS Research laboratory and University of Tuebingen in Germany [145] was undertaken to model a complex industrial application from the automotive domain, starting from high level C description down to a cycle accurate SystemC model for hardware synthesis. The aim of the project was to model the exposure control unit from the video part of the Bosch automotive driver assistance system. This case study showed that SystemC provides a helpful methodology for a seamless validation and refinement process in an industrial design flow. They have concluded that the main advantage of a SystemC-based refinement process is the fact that the specification remains executable during the design process, which means that the system can be validated at every stage of the design.

Voice and signal processor from Fujitsu

The hardware design team at Fujitsu Network Technologies Limited in Japan adopted a SystemC design flow for their latest project, voice and signal processor [146]. Their current tool-set could not handle the new multi-language requirements brought on by the addition of C/C++. They needed a verification solution that would work seamlessly, regardless of code differences. The team created a

development flow with three phases - design, verification, and synthesis. Firstly, the team produced an untimed model based on their existing C algorithm. At that point, they performed structural analysis and created a testbench to be used later in the design. Secondly, they created a functional model, in which they performed hardware allocations to optimise performance. In the next step, they created a bus cycle-accurate model using behavioural synthesis, giving them synthesisable code for the final design phase, which would create the RTL model. They were able to reuse their SystemC testbench for HDL verification. The SystemC flow reduced the verification portion of the project from two weeks to three days, and eliminated the need for hand coding and rewriting RTL. The SystemC simulations also ran approximately 150 times faster than the RTL simulations.

Ethernet adapter from Ammasso

In traditional server-to-server networking environments, administrators are faced with constant compromises between power, flexibility and cost. The team at Boston, Massachusetts based at Ammasso has developed a solution to eliminate this tradeoff [146] when creating a gigabit ethernet adapter using a 2-million-gate Xilinx Virtex-II Pro FPGA. They realised from a design perspective that a higher level methodology would be important, rather than a low level, testbench approach. They thought that writing their code in C++ rather than VHDL or Verilog would increase the efficiency and accelerate the design process. They used SystemC as their verification platform for their high level design methodology. Although the team had no prior experience with SystemC, it took them only one month to integrate the EDA Cadence verification platform into their SystemC environment.

3D graphic processor

Kogel et al [147] employed a methodology of system level design on the design of a 100 million gate 3D graphic processor. It was a large scale industrial design project. The resulting modelling efficiency measured in lines of code and simulation speed is at least two orders of magnitude better compared to an RTL model. The work was accomplished within 2 months by a team of 4 engineers familiar with the application.

Many other success stories relating to modelling using SystemC are reported on the Cadence [146] and CoWare homepages [148]. CoWare reported on their web site the feedback they received from leading systems and semiconductor companies they cooperate with, such as Infineon, Sony, InterDigital, Alcatel, STMicroelectronics and Matsushita. In all the cases, productivity increased when SystemC-based products were used in the system design flow.

On the academic side, most researchers are using SystemC to test and demonstrate its capabilities, or to add enhancements by designing a range of basic to complex systems architectures. The following are typical results reported by the academic world:

Bluetooth transceiver

Modelling of the baseband in a Bluetooth Transceiver was done in the University of Ancona, Italy to demonstrate the modelling capabilities of SystemC at a high level of abstraction [149]. Bluetooth is an emerging standard for short distance communication. The high level executable description of the baseband allowed an analysis of the behaviour of the protocol in the presence of noise and also enables a high level performance analysis. Using SystemC reduced the design time and also the typical CPU time required for the simulations.

On-chip communication AMBA

A case study of modelling an AMBA bus was presented by the University of Ancona, Italy in cooperation with STMicroelectronics in France, to demonstrate the SystemC TLM [150]. AMBA defines an on-chip communication standard for designing high performance embedded microcontrollers. The designers took advantage of SystemC communication and synchronisation features, such as channels, interfaces and events, and were able to obtain important results. Firstly, with a higher level than RTL, they gained two orders of magnitude in simulation speed. Secondly, in a bus implementation, the de-scheduling feature implemented by dynamic sensitivity allows the simulation to run faster avoiding useless function calls.

PCI bus interface

Bruschi et al [151] described functional requirements of a customised PCI bus interface. They aimed not just to develop a model but to prove modularisation, designer team cooperation, ease of modelling, possibility of reuse, comparison with VHDL-RTL model and to mix various levels of abstraction during the design flow. The developed modules include the master simulator, transceiver manager, error checker and test application. They conclude that the learning curve even for designers with little hardware design background is quite fast.

Sobel edge detection image processing algorithm

A case study of modelling a complex Sobel edge detection image processing algorithm was done by Armstrong and Ronen [152]. The case study was part of a larger effort to assess the state of the art in design tools and languages suitable for Hardware/software co-design and co-verification. The system was modelled by different SystemC structures from abstract to more concrete in terms of implementation. Their criteria in assessing an HDL are based on six aspects. It should

be able to: 1) model systems in an abstract manner but still be able to model concurrency, 2) suit either a software or hardware implementation, 3) leverage existing high level design libraries written in C, 4) support synthesis into logic or embedded processor code, 5) enjoy large industrial support and finally 6) support ease of modelling. The results of this tool and language study indicated that SystemC was a strong candidate to become the successor of VHDL and Verilog as a widely accepted modelling language and more suitable for system level modelling.

B.2 Hardware/Software Co-Design and Co-Simulation

Co-design and co-simulation are the processes of designing and simulation of systems specifications including both hardware and software. Different debugging tools are executed concurrently, to validate the algorithms and the system functionality [153].

In a traditional design methodology, hardware and software design take place in isolation with hardware being integrated with software after the hardware is fabricated [153]. Bugs that cannot be fixed in software lead to costly re-fabrication and can adversely affect time-to-market. There is a need for a system design language that describes functionality of both software and hardware [18]. The lack of a unifying system specification language has been identified as one of the main obstacles adversely affecting SoC designs. Most of the future products will be SoCs including embedded software, which now represents about 80% of a typical system.

SystemC and other languages based on C++ offer many features that can simplify the task of co-simulation [17]. Such languages offer a homogenous environment and high abstraction level of system specifications which give better component

reusability and reduce the design and verification time. Although SystemC is a powerful HDL, many issues concerning co-simulation such as embedded software generation still requires further research. Several co-simulation platforms based on SystemC have been developed by academic groups as well as EDA vendors[154, 155]. Co-simulation using SystemC is divided into two parts, homogenous and heterogeneous. A homogeneous co-simulation environment uses a single engine for simulation [154, 155, 156, 157, 158, 159], whereas, heterogeneous co-simulation environment [160, 161, 162] is based on multi-language system descriptions. Using SystemC for all the design parts will permit heterogeneous co-simulation.

Grein et al [154] presented a co-simulation environment using SystemC that provides modularity, scalability and flexibility in co-simulation of SoC designs with heterogeneous multi-processor target architectures. Modularity is achieved by modular interfaces, and scalability with easy integration of simulation models of sub-systems. Flexibility is achieved with the tradeoff between performance and accuracy. The presented environment focuses on mixed-level co-simulation between two specific abstraction levels (driver level and RTL). Experiments with an IS-95 cellular phone system design show the effectiveness of their co-simulation environment.

A co-simulation technique focusing on software generation is presented by Herrera et al [155]. This technique reduces the embedded system design cost in co-design methodology. Another main advantage is that the same SystemC code is used for system level specification and after software/hardware partition for embedded software generation. The proposed methodology is based on the redefinition and overloading of SystemC class library construction elements. In order to evaluate the proposed technique, a simple design of a car Anti-lock Braking System (ABS) is developed using a classical top-down flow methodology.

Another methodology, which attempts to enhance the support of embedded software modelling with SystemC, is SPACE (SystemC Partitioning of Architectures for Co-design of Embedded systems) [156]. In their methodology the application is partitioned into two parts, namely the software and hardware modules. Each partition can be connected to their platform via interfaces and then scheduled by the SystemC simulator. One of their contributions is that they can easily move a module from hardware to software and vice versa to allow architecture exploration.

Fummi et al [157] presented two co-simulation methodologies based on SystemC and ISS (Instruction Set Simulator) as a model of the processor. The first one works at the SystemC kernel level and exploits potentialities of the GNU suite, whereas the second uses features offered by the operating system running on the ISS. The two methodologies improve co-simulation performance with respect to the state of the art methods and provide different tradeoffs between the simplicity of the programming model, the modelling power and co-simulation performance.

The more SystemC is used for modelling hardware blocks, the more conspicuous gets the need for translating modules previously designed in HDLs into SystemC. Furthermore, heterogeneous co-simulation of multiple environments can be avoided by performing translation from one environment to another. Agliada et al [158] presented a method based on automatic translation of VHDL descriptions into SystemC with equivalent behaviour under the assumption of cycle-based simulation. They consider a simple example of a CPU (written in VHDL) and surrounding blocks (designed in SystemC and VHDL). The manual translation of the CPU into SystemC is a complex task which requires approximately 30 hours, while the automatic translation takes a few milliseconds. They have concluded that simulation time required by the SystemC only description is sensibly lower (50%) than that of VHDL simulation. The co-simulation of SystemC+VHDL is slower than SystemC-based simulation since it requires interacting of an event-driven simulator with a cycle-based one.

Another approach towards a single co-simulation environment based on translating Verilog to SystemC [159] has been proposed by Mahmoudi et al The researchers claim that their conversion methodology covers the synthesisable subset of Verilog. Almost for all Verilog constructs, there is a SystemC equivalent except for delays, dynamic constructs and wait statements. A comparison between simulation speeds of SystemC and Verilog shows that there is a 27% speed up factor in SystemC simulation times over those of Verilog.

Heterogeneous SystemC-VHDL co-simulation is presented by Bombana and Bruschi [160]. The task covers the possibility of modelling the application at RTL and/or behavioural SystemC level, mixing VHDL and SystemC modules both in the model itself and in the testbench. Also, the feasibility of applying synthesis, and the possibility of mixed representation co-simulation is demonstrated.

Yuyama et al [161] proposed a co-design methodology using SystemC and a high level synthesis tool named BachC [56]. The hardware part is implemented to a peripheral block through BachC, while the software part is converted to an embedded software on the CPU. BachC is a C-based high level synthesis system. They developed a SystemC library to connect SystemC to BachC, and applied their methodology to JPEG encoder including an embedded CPU and a peripheral block. The peripheral block model is automatically translated to BachC. Comparing area and performance between BachC models and a hand-coded Verilog RTL model, the area from BachC is larger but its throughput can be better than that from the Verilog RTL model. The structure of the Verilog RTL model is fixed, while VHDL RTL models generated from the Bach system can be flexibly varied according to a given clock frequency.

A top-down design methodology from C to silicon is proposed by Cai et al [162]. They have chosen SpecC [20], VCC [163] and SystemC as a modelling environment because they are all C-related and each has a strong support in at least one

design field. The methodology combines the design flows of SpecC and VCC with SystemC added as a back-end to them. This method quickly converts the C model to an implementation resulting in a decreased design cycle time.

In a short period of time the SystemC popularity has generated a lot of support from the EDA industry. This is obvious from the rich collection of SystemC simulators. e.g. ConvergenSC System design from Coware [164], BlueHDL from Blue Pacific [25] and the tri-lingual ModelSim SE V6.0 from Mentor Graphics [165]. All these simulators are compliant with OSCI SystemC. Most of them can be used in conjunction with other languages and provide textual as well as graphical views of SystemC designs.

B.3 Co-Verification

Co-verification is the process used to demonstrate the correctness of a design consisting of hardware and software. A big part of the design effort is dedicated to verification, especially in complex SoC designs [166]. Every time a design description at an identified abstraction level is converted to a description at a lower level, it is necessary to run a verification phase. Innovation in tools and methodology are needed to make the verification process faster and more precise. Verification tools enable the functionality, performance and testability of a design which needs to be evaluated prior to fabrication. This is essential in ensuring that the design will work first time and will remain robust and resilient in service under all operating conditions.

The SystemC standard contains enough features for an effective verification of a real design [167]. This attracts the EDA vendors and researchers towards SystemC. Their contributions cover most of the verification options, mainly formal techniques [168, 169] and simulation based techniques. Simulation based techniques

include the design to validate examples explained in section B.2, co-verification, transaction based simulations, AMS simulations which will be introduced later in section 2.3, and hardware based tools such as emulation systems, rapid prototype systems and hardware accelerators.

Formal verification approaches use rigorous mathematical reasoning to show that a design meets all or parts of its specification [168]. The first example of SystemC based formal verification was introduced by Grobe and Drechsler [169] which allows proving the correctness of properties specified in linear temporal logic.

Based on system level features in SystemC, the Transaction Based Verification (TBV) methodology is designed to raise the level of abstraction so that it is easy to create and reuse testbenches, easy to debug and run simulations. Based on this idea, there have been many SystemC TBV environments [170, 171, 167, 172, 173], but the main contribution was from Cadence Design Systems verification engineers [170, 171] who created an environment called TestBuilder. TestBuilder supports functional verification, fills several missing pieces in SystemC 2.0 and acts as a verification layer on top of SystemC. TestBuilder has been accepted by the SystemC Steering Committee as a standard.

Other co-verification environments [174, 175, 176] illustrate the benefits of using the same language for hardware and software design of a single system, enabling the development of very fast models at various levels of abstraction.

Most of the SystemC verification approaches are based on the definition of fault models [177, 178, 179, 180] that take advantage of the transaction level and have the ability to use VHDL or Verilog. For instance, AMELTO and LAERTE++ [177, 178] from the University of Verona, Italy are multi-language environments developed to efficiently test embedded systems and IP cores.

A new approach of integrating SystemC with hardware based verification was introduced by Ramaswamy and Tessier [181]. Their approach represents an integration of SystemC and the IKOS Virtualogic emulation system which is a parallel logic verification tool, to improve the verification performance while maintaining verification fidelity across a range of abstraction levels.

A SystemC verification group was formed on November 2001 aiming to explore infrastructure and methodology for using SystemC for functional verification. The first outcome from the group was the release of the SystemC Verification library (SCV) 1.0 in December 2003. SCV 1.0 [182] has been approved as an official OSCI standard and is available for download from the OSCI homepage. SCV is based on TestBuilder from Cadence Design Systems [146]. SCV provides a C++ signal class, which interfaces C++ to an HDL design at the signal level. SCV supports abstraction of tests to the transaction level. It provides a powerful randomisation facility, including multiple constrained random generators (integer, float, signal, or custom) that are able to execute simultaneously. SCV provides event expressions, the enabling technology for creating temporal expressions, monitors, and temporal checks. SCV supports both Verilog and VHDL. Since its release SCV has been widely used by the design community and was subject to further enhancements [183].

There are several commercial SystemC verification tools introduced by the EDA industry, such as Seamless from Mentor Graphics [35], CoCentric System Studio from Synopsys [26], N2C from CoWare [184], Nexus-PDK from Celoxica [185], and Visual Elite from Summit Design [186].

B.4 Synthesis

A synthesis process aims at finding the best equivalent representation in the next level of abstraction which guarantees the same functionality as the design in the current level of abstraction considering the constraints given by the designer [187]. There are three common synthesis steps: behavioural synthesis, logic synthesis and physical synthesis. They transform the initial specifications from behavioural to RTL, then from RTL to gate level and finally from gate level to layout respectively.

In the traditional design methodology (see Section 1.1) system level designers typically use C/C++ based development environments to specify systems. Then hardware designers manually translate the executable specifications from C/C++ into HDL and continue adding more detail until the HDL code can be synthesised into a gate level netlist [188].

One of the problems with this methodology is that rewriting a C/C++ code into an equivalent HDL description is both time consuming and error prone. What is needed is a smooth and reliable methodology that allows the hardware designer to continue refining the C/C++ executable specification into a form that is acceptable as input for hardware synthesis, without the need to translate the C/C++ code into an HDL. Eliminating the translation step enables the reuse of the original C/C++ testbench, decreases the verification time and ensures compliance with the original specification.

SystemC was not intentionally developed to create executable specifications of hardware components when aiming at synthesis. This used to be one of the most important disadvantages of SystemC when it was first announced. To overcome this problem designers tried to provide automatic hardware synthesis frameworks to the SystemC community [142, 143].

A framework called SystemC-Plus [189] was developed under the European Commission's project ODETTE. It is completely based on SystemC and provides synthesisable object-oriented features. It translates an object-oriented input description into a description that can be processed by existing logic synthesis tools but it will not directly produce a gate level netlist. Another framework [190] performs high level synthesis by taking SystemC behavioural input specifications to generate VHDL, Verilog or SystemC RTL output specifications.

The EDA industry introduced many SystemC based synthesis tools, for examples CoCentric SystemC compiler from Synopsys [26] and Cynthesizer from Forte Design [27]. These tools are able to synthesise hardware descriptions written in SystemC into RTL or gate level netlists. EDA synthesis tools received a great attention from the designers, to the extent that designers have tried to detect weaknesses and cure them [191, 192].

B.5 Further Enhancement and Extensions

Visualisations of designs described in SystemC are now available at system level [193, 194, 195]. Designers are getting interested in tools for SystemC that can easily describe systems and keep a global view of them. Difficulties for designers arise from the low abstraction level of results provided by a SystemC simulation. Designers can obtain a "*.vcd" file (results file) which may be displayed by freeware viewers. Whatever tool is used to display simulation waveforms, only information about variables or signals can be obtained, but not on processes activity or synchronisation between modules. Some authors have proposed to add a Graphical User Interface (GUI) to SystemC [193] based on a graphical view of an object-oriented description of the application. This requires a modification to the SystemC kernel. CoCentric introduced a tool [26] to graphically display complex

data structures, while Moigne [194] went further and introduced a tool to display synchronisation with events.

The evolution of SystemC from V0.9 to V2.0.1 suggests that the environment is particularly geared towards framework functionalities and performance. In each new version, more libraries were added for communication methodologies and interfaces between modules [196, 197]. Furthermore, methods are described for system modelling to cover a wide range of MoC [198]. Improvements have been suggested to speed up the SystemC engine e.g. by proposing new scheduling techniques [199].

The concept of overloading operators in C++ easily suggests extensions to the language by adding new data types and mixing them with the native SystemC. For instance, a multi-valued logic to model and simulate multi-valued circuits can be added quite easily [200].

Another extension was suggested to allow performance evaluation at system level to avoid costly iterations in the design process. Such performance evaluation relies on timing properties (execution times, delays, periods, etc.) which are important especially in the performance verification of multiprocessing [201].

Existing simulation models that have been used for a reasonable amount of time and were iteratively improved over some generations of implementations are likely to be thrown away when changing to a new level or language. Methods are suggested for an easy migration of C-models to SystemC-based designs e.g. by using global variables and interfaces [202].

Appendix C

SystemC-A Models

This appendix presents all SystemC-A models developed in this project which were not included within the context of this thesis's chapters. Section C.1 provides the listings of circuit-level components, while Section C.2 presents the remainder of modules of the PLL described in Section 6.4.

C.1 Circuit-Level Components

Circuit-level components described in this section are resistor, capacitor, diode, MOSFET transistor, DC voltage source, sinusoidal voltage source, ramp voltage source and a sinusoidal current source.

C.1.1 Resistor sc_a_resistor

```
1  #include "sc_a_resistor.h"
2  ...
3  // resistor interface contains name, 2 nodes and a resistance
4  sc_a_resistor::sc_a_resistor(char nameC[5], sc_a_system_variable *node_a,
5  sc_a_system_variable *node_b, double value):
6     sc_a_component(nameC, node_a, node_b, value){}
7
8  void sc_a_resistor::BuildM(void){
9     G=1/value;
10    Jacobian(a,a,G);
```

```
11      Jacobian(a,b,-G);
12      Jacobian(b,a,-G);
13      Jacobian(b,b,G);
14  }
15
16  void sc_a_resistor::BuildB(void){
17      G=1/value;
18      double i=(X(a)-X(b))*G;
19      BuildRhs(a,-i);
20      BuildRhs(b,i);
21  }
```

LISTING C.1: SystemC-A model of a resistor.

C.1.2 Capacitor sc_a_capacitor

```
1   #include "sc_a_capacitor.h"
2   ...
3   // capacitor interface contains name, 2 nodes, capacitance and initial
4   // capacitor voltage
5   sc_a_capacitor::sc_a_capacitor(char nameC[5],sc_a_system_variable *node_a,
6   sc_a_system_variable *node_b,double value,double Vc):
7   sc_a_component(nameC,node_a, node_b, value){
8       Vc0=Vc;
9   }
10
11  void sc_a_capacitor::BuildM(void){
12      C=value;
13      S=S();
14
15      Jacobian(a,a,S*C);
16      Jacobian(a,b,-S*C);
17      Jacobian(b,a,-S*C);
18      Jacobian(b,b,S*C);
19  }
20
21  void sc_a_capacitor::BuildB(void){
22      C=value;
23      Vdotn=Xdot(a)-Xdot(b);
24
25      BuildRhs(a,-C*Vdotn);
26      BuildRhs(b,C*Vdotn);
27  }
```

LISTING C.2: SystemC-A model of a capacitor.

C.1.3 Diode sc_a_diode

```
1   #include "sc_a_diode.h"
2   ...
3   // diode interface contains name, 2 nodes, Gdmax, lamda and saturation current.
4   sc_a_diode::sc_a_diode(char nameC[5],sc_a_system_variable *node_a,
5   sc_a_system_variable *node_b,double value,double lamda,double Is):
6   sc_a_component(nameC,node_a, node_b, value){
7       Gdmax=value;//diode max conductance
8       lam=lamda;
9       satc=Is;
10  }
11
```

```
12  void sc_a_diode::BuildM(void){
13      vdm=X(a)-X(b);
14      Vdmax=log(Gdmax/(lam*satc))/lam;// calculate diode max voltage
15      if (vdm>Vdmax){//use straight line model
16          Idmax=satc*exp(lam*Vdmax);
17          Gd=lam*Idmax;
18      }
19      else            //use exponential model
20      Gd=lam*satc*exp(lam*vdm);
21
22      Jacobian(a,a,Gd);
23      Jacobian(a,b,-Gd);
24      Jacobian(b,a,-Gd);
25      Jacobian(b,b,Gd);
26  }
27
28  void sc_a_diode::BuildB(void){
29      vdm=X(a)-X(b);
30      Vdmax=log(Gdmax/(lam*satc))/lam;
31      if (vdm>Vdmax){//use straight line model
32          Idmax=satc*exp(lam*Vdmax);
33          Id=Idmax+lam*Idmax*(vdm-Vdmax);
34      }
35      else                //use exponential model
36      Id=satc*(exp(lam*vdm)-1);
37
38      BuildRhs(a,-Id);
39      BuildRhs(b,Id);
40  }
```

LISTING C.3: SystemC-A model of a diode.

C.1.4 MOSFET sc_a_mosfet

```
1   #include "sc_a_mosfet.h"
2   ...
3   // mosfet interface consists of name, 3 nodes, VT, K, Gdsmin
4   sc_a_mosfet::sc_a_mosfet(char nameC[5],sc_a_system_variable
5   *node_a,sc_a_system_variable *node_b,sc_a_system_variable *node_c,double VT1,
6   double K1, double Gdsmin1):
7   sc_a_component(nameC,node_a, node_b, value){
8      c=node_c;
9      VT=VT1;// gate threshold voltage
10     K=K1;// gain
11     Gdsmin=Gdsmin1;// defined to help the solver not to fail numerically
12  }
13
14  void sc_a_mosfet::BuildM(void){
15     vd=X(a);
16     vs=X(c);
17     vg=X(b);
18     gds=ggs=Ids=0;
19     if (vd>vs){
20         vds=vd-vs;
21         vgs=vg-vs;
22     }
23     else{
24         vds=vs-vd;
25         vgs=vg-vd;
26     }
27     vgst=vgs-VT;
28     double Gl=1e-3;//to solve problems in nonlinear solver
29         if (vgst<=0){//cut off
30             gds=0.0;
31             ggs=0.0;
```

```
32          }
33          else if (vds<vgst){// linear
34                  gds=K*(vgst-vds)+ Gl;
35                  ggs=K*vds;
36          }
37          else{//saturation
38                  gds=Gl;
39                  ggs=K*vgst;
40          }
41
42          if(vd<vs)//other direction
43              Ids=-Ids;
44
45      //a drain, b gate, c source
46      Jacobian(a,a,gds+Gdsmin);
47      Jacobian(a,b,ggs);
48      Jacobian(a,c,-gds-Gdsmin-ggs);
49      Jacobian(c,a,-gds-Gdsmin);
50      Jacobian(c,b,-ggs);
51      Jacobian(c,c,gds+Gdsmin+ggs);
52  }
53
54  void sc_a_mosfet::BuildB(void){
55      vd=X(a);
56      vs=X(c);
57      vg=X(b);
58      gds=ggs=Ids=0;
59
60      if (vd>vs){
61          vds=vd-vs;
62          vgs=vg-vs;
63      }
64      else{
65          vds=vs-vd;
66          vgs=vg-vd;
67      }
68          vgst=vgs-VT;
69
70      double Gl=1e-3;
71          if (vgst<=0) //cut off
72              Ids=0.0;
73          else if (vds<vgst)//linear
74              Ids=K*(vgst*vds-vds*vds*0.5)+vds*Gl;
75          else                //saturation
76              Ids=0.5*K*vgst*vgst+vds*Gl;
77
78          if(vd<vs)//other direction
79              Ids=-Ids;
80
81      //a drain, b gate, c source
82      BuildRhs(a,-Ids);
83      BuildRhs(c,Ids);
84  }
```

LISTING C.4: SystemC-A model of a MOSFET transistor.

C.1.5 DC Voltage Source sc_a_voltageS_dc

```
1   #include "sc_a_flow.h"
2   #include "sc_a_voltageS_dc.h"
3   ...
4   //dc voltage source interfaces contain name, 2 nodes and a value in volts
5   sc_a_voltageS_dc::sc_a_voltageS_dc(char nameC[5],sc_a_system_variable *node_a,
6   sc_a_system_variable *node_b,double value):
7   sc_a_component(nameC,node_a, node_b, value){
```

```
8       i1 = new sc_a_flow("i1");//according to MNA define flow variable
9   }
10
11  void sc_a_voltageS_dc::BuildM(void){
12      Jacobian(a,i1,-1);
13      Jacobian(b,i1,1);
14      Jacobian(i1,a,1);
15      Jacobian(i1,b,-1);
16  }
17
18  void sc_a_voltageS_dc::BuildB(void){
19      i=X(i1);
20      vab=X(a)-X(b);
21      E=value;
22
23      BuildRhs(i1,E-vab);//subtract v from node voltage (E-v1)
24      BuildRhs(a,i);
25      BuildRhs(b,-i);//subtract I from RHS (v*G-I)
26  }
```

LISTING C.5: SystemC-A model of a DC voltage source.

C.1.6 Sine Wave Voltage Source sc_a_voltageS_sin

```
1   #include "sc_a_voltageS_sin.h"
2   ...
3   //sin wave voltage source interface contains name, 2 nodes and sine
4   //wave generic values.
5   sc_a_voltageS_sin::sc_a_voltageS_sin(char nameC[5],sc_a_system_variable *node_a,
6   sc_a_system_variable *node_b,double value,double Voffset,double Amplitude,
7   double Delay, double Damping):
8   sc_a_voltageS(nameC,node_a, node_b, value){
9       freq=value;// sine wave generic values
10      V_off=Voffset;
11      Amp=Amplitude;
12      TD=Delay;
13      Theta=Damping;
14  }
15
16  void sc_a_voltageS_sin::BuildM(void){
17      Jacobian(a,i1,-1);
18      Jacobian(b,i1,1);
19      Jacobian(i1,a,1);
20      Jacobian(i1,b,-1);
21  }
22
23  void sc_a_voltageS_sin::BuildB(void){
24      I=X(i1);
25      t=get_time();
26      value=V_off+Amp*sin(2*3.14*freq*(t-TD))*exp(-Theta*(t-TD));
27      Va=X(a);
28
29      BuildRhs(i1,value-Va);
30      BuildRhs(a,I);
31      BuildRhs(b,-I);
32  }
```

LISTING C.6: SystemC-A model of a sine wave voltage source.

C.1.7 Current Source sc_a_currentS_dc

```
1  #include "sc_a_currentS_dc.h"
2  ...
3  sc_a_currentS_dc::sc_a_currentS_dc(char nameC[5],sc_a_system_variable *node_a,
4  sc_a_system_variable *node_b, const double *value):
5  sc_a_component(nameC,node_a, node_b, *value){
6      Ivalue=value;
7  }
8
9  void sc_a_currentS_dc::BuildB(void){
10     Equation(a,*Ivalue);
11     Equation(b,-(*Ivalue));
12 }
```

LISTING C.7: SystemC-A model of a DC current source.

C.2 Phase Locked Loop

This section presents the different modules of the PLL case study modelled in Section 6.4. The modules are the detector, charge pump and filter, and divide by N.

C.2.1 Detector

```
1  #include "systemc.h"
2
3  SC_MODULE(detector){
4      sc_in<bool> Ref1, DivVco1;//inputs: reference and divider signals
5      sc_out<bool> OutQ1, OutQ2;// outputs: Q1 & Q2
6
7   void DET1();
8   bool Q1, Q2;
9
10 SC_CTOR(detector){
11         SC_METHOD(DET1);
12         dont_initialize();
13         sensitive_pos << Ref1<< DivVco1;
14         Q1 =false;
15         Q2 =false;
16     }
17 };
18
19 void detector::DET1(){
20     bool clear;
21     clear=false;
22     //modelling NAND gate
23     if ( Ref1.event() && Q2)
24         clear=true;
25
26     if ( DivVco1.event() && Q1)
27         clear=true;
28
```

```
29      // modelling D flip flop
30      if ( clear )//if clear is true the two outputs are false
31      {
32          Q1=false;
33          Q2=false;
34      }
35
36      else {
37          if( Ref1.event() )
38              Q1=true;
39
40          if( DivVco1.event() )
41              Q2=true;
42      }
43      OutQ1.write(Q1);
44      OutQ2.write(Q2);
45  }
```

LISTING C.8: SystemC-A model of the detector in PLL.

C.2.2 Charge Pump and Filter

```
1   #include "systemc.h"
2   #include "sc_a_node.h"
3
4   //integrator and lead/lag low pass filter
5   SC_MODULE(filter){
6       sc_in<bool> clk;//to enforce a fixed time step
7       sc_in<bool> OutQ1, OutQ2;//inputs: Q1&Q2 from detector's output
8       sc_out<double> Iout;//output: charge pump current
9
10      void Vf();
11      double Vc1, Vc2, C1, C2, R, h, Iin;
12      void init();
13      Node *n0,*n1,*n2;
14
15      SC_CTOR(filter){
16          SC_METHOD(Vf);
17          dont_initialize();
18          sensitive_pos << clk;//filter should be sensitive to clk to get noise signal
19                                // correctly
20          init();
21          //filter components values
22          Vc1=Vc2=0.0;
23          C1=3e-9;
24          C2=4000e-12;
25          R=1e3;
26          }
27  };
28
29  void filter::init(){//filter modelled at circuit level
30      n2=new sc_a_node("n2");
31      n0=new sc_a_node("0");
32      n1=new sc_a_node("n1");
33      sc_a_currentS_dc *I1=new sc_a_currentS_dc("I1", n1,n0,&Iin);
34      sc_a_capacitor *c1=new sc_a_capacitor("c1",n1,n2,3e-9,1);
35      sc_a_resistor *r1=new sc_a_resistor("r1",n2,n0,1e3);
36      sc_a_capacitor *c2=new sc_a_capacitor("c2",n2,n0,4e-9,1);
37  }
38
39  void filter::Vf(){
40      //charge pump
41      if (OutQ1.read())
42          Iin=500e-6;
43      else if (OutQ2.read())
```

```
44          Iin=-500e-6;
45      else
46          Iin=0;
47
48      Iout.write(Iin);
49  }
```

LISTING C.9: SystemC-A model of the charge pump and filter in PLL.

C.2.3 Divide by N

```
1   #include "systemc.h"
2
3   SC_MODULE(DivideByN){
4       sc_in<bool> Vco;//input: from vco
5       sc_out<bool> DivVco;//output: divider signal
6
7       double t1,t2,width, jitter;
8       int count1;
9       bool V;
10      void Counter();
11
12  SC_CTOR(DivideByN){
13          SC_METHOD(Counter);
14          dont_initialize();//prevent run at vco=0
15          sensitive_pos << Vco;
16          count1=0, V=0, t2=0, t1=0, width=0, jitter=0;
17  }
18  };
19
20  void DivideByN::Counter(){
21      if (Vco.event()){
22          if (++count1>=2000){// divider ratio N=2000
23              count1=0;
24              V=!DivVco.read();
25          }
26
27          t1=sc_time_stamp().to_seconds();
28          width=t1-t2;
29          jitter=width-2.5e-10;//calculate jitter w.r.t 2.5e-10 the
30                               //true signal width
31          t2=t1;
32      }
33      DivVco.write(V);
34  }
```

LISTING C.10: SystemC-A model of the divide by N in PLL.

Appendix D

VHDL-AMS Models

This appendix lists VHDL-AMS models that are developed within the work of this project for comparisons with SystemC-A models. Section D.1 lists all the PLL's modules, while Section D.2 and D.3 present the original and proposed Jiles-Atherton ferromagnetic hysteresis models. For VHDL-AMS models of the automotive vibration isolation system please refer to [141].

D.1 Phase Locked Loop

This section presents the different modules of the PLL case study modelled for comaprision with SystemC-A model in Section 6.4. The modules are the detector, charge pump, filter, divide by N, VCO, and the testbench.

D.1.1 Detector

```
1  library IEEE;
2  use ieee.std_logic_1164.all;
3
4  -- digital phase detector
5  entity PhaseDetector is
6   port( Ref,Div :in std_logic; -- inputs
7         OutQ1, OutQ2 :out std_logic -- outputs
```

```
8       );
9   end entity PhaseDetector;
10
11  architecture Structure of PhaseDetector is
12  begin
13      process (Ref, Div)
14          variable clear: bit;
15          variable Q1, Q2: std_logic:='0';
16
17          begin
18          -- AND gate model
19              clear:='0';
20              if Ref='1' and Q2='1' then
21                  clear :='1';
22              end if;
23              if Div='1' and Q1='1' then
24                  clear := '1';
25              end if;
26              -- D flip flop model
27              if clear = '1' then
28                  Q1 := '0';
29                  Q2 := '0';
30              else
31                  if Ref='1' then
32                      Q1 := '1';
33                  end if;
34                  if Div='1' then
35                      Q2 := '1';
36                  end if;
37              end if;
38
39              OutQ1 <= Q1;
40              OutQ2 <= Q2;
41      end process;
42  end architecture Structure;
```

LISTING D.1: VHDL-AMS model of the detector in PLL.

D.1.2 Charge Pump

```
1   library IEEE;
2   use ieee.std_logic_1164.all;
3   use ieee.electrical_systems.all;
4   use IEEE.math_real.all;
5
6   -- charge pump
7   entity CP is
8    port(OutQ1, OutQ2 :in std_logic;      -- digital inputs
9           terminal Tpump :electrical); -- analogue output
10  end entity CP;
11
12  architecture behav of CP is
13      quantity  vv across Ic through ELECTRICAL_REF to Tpump;
14      signal Iin :real:=0.0;
15      begin
16      process (OutQ1, OutQ2)
17      begin
18          if OutQ1 ='1' then
19              Iin <= 0.0005;
20
21          elsif OutQ2 = '1' then
22              Iin <= -0.0005;
23          else
24              Iin <= 0.0;
25          end if;
```

```
26      end process;
27      Ic==Iin'ramp;
28  end architecture behav;
```

LISTING D.2: VHDL-AMS model of the charge pump in PLL.

D.1.3 Filter

```
1   LIBRARY ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.electrical_systems.all;
4
5   LIBRARY edulib;
6   use work.all;
7
8   -- integrator and lead/lag low pass filter
9   entity CIRCUIT is
10      port (terminal n1T,n2T: ELECTRICAL);
11  end entity CIRCUIT;
12
13  architecture arch_CIRCUIT of CIRCUIT is
14      component RESISTOR
15          generic( RES : RESISTANCE );
16          port( terminal P1 :  ELECTRICAL;
17                terminal P2 :  ELECTRICAL );
18      end component RESISTOR;
19
20      component CAPACITOR
21          generic( CAP : CAPACITANCE;
22                   V_IC : REAL:=REAL'LOW );
23          port( terminal P1 :  ELECTRICAL;
24                terminal P2 :  ELECTRICAL );
25      end component CAPACITOR;
26       -- use components from System Vision Components library
27      for R1: RESISTOR use entity EDULIB.RESISTOR(IDEAL);
28      for C1: CAPACITOR use entity EDULIB.CAPACITOR(IDEAL);
29      for C2: CAPACITOR use entity EDULIB.CAPACITOR(IDEAL);
30
31  begin
32      R1 : RESISTOR
33          generic map ( RES => 1000.0 )
34          port map ( P1 => ELECTRICAL_REF,
35                     P2 => n2T );
36
37      C1 : CAPACITOR
38          generic map ( CAP => 3.0E-9 )
39          port map ( P1 => n1T,
40                     P2 => n2T );
41      C2 : CAPACITOR
42          generic map ( CAP => 4.0E-9 )
43          port map ( P1 => n2T,
44                     P2 => ELECTRICAL_REF );
45
46  end architecture arch_CIRCUIT;
```

LISTING D.3: VHDL-AMS model of the filter in PLL.

D.1.4 Divide by N

```
1  library IEEE;
2  use ieee.std_logic_1164.all;
3
4  -- Divider
5  entity dividebyN is
6   port( Vco :in std_logic;        --input
7           DivVco :out std_logic  --output
8       );
9  end entity dividebyN;
10
11 architecture behav of dividebyN is
12      begin
13      process (Vco)
14          variable count :integer:=0;
15          variable Divide:std_logic:='0';
16          variable N:integer:=2000; --divide by 2000
17      begin
18          DivVco<= Divide;
19          if Vco = '1' then
20              count:=count+1;
21              if count >= N then
22                  count:=0;
23                  Divide:= not Divide;
24              end if;
25          end if;
26      end process;
27 end architecture behav;
```

LISTING D.4: VHDL-AMS model of the divide by N in PLL.

D.1.5 VCO

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.electrical_systems.all;
4  use IEEE.math_real.all;
5
6  entity -- modified VCO version of Ashenden book vco is
7      generic (fmax    : real := 5.0e9;     -- Frequency when input voltage = vmax
8               fmin    : real := 2.0e9;     -- Frequency when input voltage = vmin
9               vmax    : real := 3.3;       -- Input voltage for fmax
10              vmin    : real := 0.0);      -- Input voltage for fmin
11     port (terminal v_input : electrical; -- analogue input
12         d_out   : out std_logic);          -- digital output
13 end entity vco;
14
15 architecture behavioral of vco is
16   quantity period : real;
17   quantity v across v_input to electrical_ref;
18   constant gain : real := (fmax - fmin)/(vmax - vmin);-- VCO gain
19 begin
20   process
21   begin
22     d_out <= '0';
23     wait until domain = time_domain;
24     loop
25       d_out <= '0';
26       wait for period/2.0;
27       d_out <= '1';
28       wait for period/2.0;
29     end loop;
```

```
30    end process;
31
32    period == 1.0/((v - vmin)*gain + fmin);
33  end architecture behavioral;
```

LISTING D.5: VHDL-AMS model of the VCO in PLL.

D.1.6 Testbench

```
1   library IEEE;
2   USE ieee.electrical_systems.all;
3   use IEEE.math_real.all;
4   use ieee.std_logic_1164.all;
5
6   -- Phase locked loop testbench
7   entity PLL is
8    port( Input :in std_logic);
9   end entity PLL;
10
11  architecture MixedSignal of PLL is
12   signal  DivT:std_logic:='0';
13   signal Up,Down,VCO_out : std_logic:='0';
14   terminal CP_Filter,Filter_VCO :electrical;
15
16   component PhaseDetector is
17    port( Ref,Div :in std_logic; -- inputs
18          OutQ1, OutQ2 :out std_logic -- outputs
19        );
20   end component;
21
22   component CP is
23   port( OutQ1,OutQ2 :in std_logic;
24         terminal Tpump :electrical
25       );
26   end component;
27
28   component circuit  is
29    port (terminal n1T,n2T: ELECTRICAL);
30   end component;
31
32   component VCO is
33     generic (fmax   : real := 5.0e9;
34              fmin   : real := 2.0e9;
35              vmax   : real := 3.3;
36              vmin   : real := 0.0);
37
38    port ( terminal v_input :electrical;
39           d_out   : out std_logic);
40   end component;
41
42   component dividebyN is
43      port( Vco :in std_logic;
44            DivVco :out std_logic);
45   end component;
46
47  begin
48    detector0: PhaseDetector
49       port map (Ref => Input,
50                 Div => DivT,
51                 OutQ1  => Up,
52                 OutQ2 => Down);
53
54    charge_pump:CP
55      port map (OutQ1=>Up,
56                OutQ2=>Down,
```

```
57                      Tpump=>CP_Filter);
58
59    Filter: Circuit
60       port map (n1T =>CP_Filter ,
61                 n2T => Filter_Vco);
62    VCO0: VCO
63     generic map(fmax => 5.0e9,
64              fmin    => 2.0e9,
65              vmax    => 3.3,
66              vmin    => 0.0)
67
68       port map ( v_input => Filter_Vco,
69                        d_out=>VCO_out);
70
71    Divider:dividebyN
72       port map ( vco => VCO_out,
73                  DivVco => DivT);
74
75 end architecture MixedSignal;
```

LISTING D.6: VHDL-AMS testbench of the PLL.

D.2 Original Jiles-Atherton Model

This model is written by Wilson et al [126].

```
1
2  Library IEEE;
3  use ieee.electrical_systems.all;
4  use IEEE.math_real.all;
5
6    entity core2 is
7      generic (k,c,ms,alpha,a,area:real);
8      port (terminal p,m :magnetic;
9          quantity B: out real);
10   end entity core2;
11
12 ARCHITECTURE core2_ja OF core2 IS
13    CONSTANT MU0:real:=4.0e-7*MATH_PI;
14    CONSTANT mg:real:=MU0*area;
15    QUANTITY h across flux through p TO m;
16    QUANTITY he,dhdt,dm,dmdh,dmdhl,dmirrdt,mirr,mirrcalc:real:=0.0;
17    QUANTITY man,mtotal,mrev:real:=0.0;
18    QUANTITY delta:real:=1.0;
19
20    FUNCTION lang (x : real) RETURN real is --Langiven's function
21     variable lang_x :real := 0.0;
22     BEGIN
23          If abs(x) < 1.0e-3 then
24              Lang_x:=0.333*x;
25          Else
26              Lang_x:=1.0/tanh(x) -1.0/x;
27          End if;
28              RETURN lang_x;
29     END FUNCTION;
30
31 BEGIN
32      -- calculate he and derivative of h
33      he == h + (alpha * ms * mtotal);
34      dhdt==he'dot;
35
36      -- Get the field direction
```

```
37      IF dhdt > 0.0 USE
38          delta == 1.0;
39      ELSE
40          delta == -1.0;
41      END USE;
42
43      -- Anhysteretic Magnetization
44      man == lang(he/a);
45      mrev == c * man / ( 1.0 + c );
46
47      -- Calculate incremental Magnetization
48      dm==man-mtotal;
49
50      -- calculate dM/dH and perform limitation on it
51      dmdh1== dm/(delta*k - alpha*ms*dm);
52      if dmdh1>0.0 use
53          dmdh==dmdh1;
54      else
55          dmdh==0.0;
56      end use;
57
58      -- calculate dM/dt and then integrate to get Mirr
59      dmirrdt==dhdt*dmdh;
60      mirrcalc==dmirrdt'integ;
61      mirr == 1.0 * mirrcalc / ( 1.0 + c );
62
63      -- Calculate Total Magnetization
64      mtotal == mrev + mirr;
65
66      -- Calculate Flux and Flux Density
67      flux == mg * (ms * mtotal + h);
68      B == flux/area;
69
70  END ARCHITECTURE core2_ja;
```

LISTING D.7: VHDL-AMS implementation of the original Jiles-Atherton ferromagnetic hysteresis model.

D.3 Proposed Jiles-Atherton Model

```
1   library disciplines;
2   use disciplines.electromagnetic_system.all;
3
4     entity core1_ja is
5       generic (k,c,ms,alpha,a,area:real);
6       port (terminal p,m :magnetic);
7     end entity core1_ja;
8
9     ARCHITECTURE core1_ja OF core1 IS
10      CONSTANT MU0:real:=4.0e-7*MATH_PI;
11      CONSTANT mg:real:=MU0*area;
12      CONSTANT dhmax:real:=12.0;
13      QUANTITY H across flux through p TO m;
14      QUANTITY He,B,mrev,mirr,mtotal,man:real:=0.0;
15      SIGNAL lasth,deltah,mirrsig:real:=0.0;
16      SIGNAL hchanged, trig: boolean:=false;
17
18      FUNCTION lang_mod(x : real) RETURN real is --Langiven's function
19       variable lang_x :real := 0.0;
20       BEGIN
21           lang_x := (2.0/MATH_PI)*arctan(x);
22          RETURN lang_x;
23      END FUNCTION;
```

```
24
25    BEGIN
26    -- hchanged signal assignment triggered by sufficient changes in field strength
27    hchanged <= H'above(lasth+dhmax) or not H'above(lasth-dhmax);
28
29      -- Simultaneous statement to calculate He
30      He == H + (alpha * ms * mtotal);
31
32      -- Anhysteretic Magnetization
33      man == lang_mod(He/a);
34      mrev ==  c * man / ( 1.0 + c );
35
36      -- Calculate Total Magnetization
37      mirr == mirrsig;
38      mtotal == mrev + mirrsig;
39
40      -- Calculate Flux and Flux Density
41      flux == mg * ( ms * mtotal + H);
42      B == flux/area;
43
44    --process to monitor H triggered by hchanged
45      PROCESS (hchanged)IS
46         VARIABLE dh : real := 0.0;
47      BEGIN
48           trig<=false;
49           dh := (H-lasth);
50           if abs(dh) > dhmax then
51              deltah <= dh;
52              lasth <= H;
53              trig<=true;
54           end if;
55      END PROCESS;
56
57      -- process to integrate dM/dH with Euler method
58      PROCESS (trig) IS
59           variable dk : real := 0.0;
60           variable deltam,dm,dmdh,dmdh1,dh: real;
61      BEGIN
62      if deltah > 0.0 then -- get field direction
63               dk:=k;-- rising
64           else
65               dk:= -k; --falling
66           end if;
67
68      -- Forward Euler integration method
69      dh := deltah;
70      deltam:= man - mtotal;
71      dmdh1 := deltam/((1.0 +c)*(dk - (alpha*ms*deltam)));
72      if dmdh1>0.0 then
73          dmdh:=dmdh1;
74      else
75          dmdh:=0.0;
76      end if;
77
78      dm:=dh*dmdh;
79
80      if dh * dm < 0.0 then
81          dm:=0.0
82      end if;
83
84      -- JA Model
85      mirrsig <=  mirrsig + dm;
86
87    END PROCESS;
88  END ARCHITECTURE core1_ja;
```

LISTING D.8: VHDL-AMS implementation of the proposed Jiles-Atherton ferromagnetic hysteresis model.

References

[1] M. Hamour, R. Saleh, S. Mirabbasi, and A. Ivanov, "Analog IP Design Flow for SoC Applications," in *Proceedings International Symposium on Circuits and Systems*, Bangkok, Thailand, 25-28 May 2003.

[2] A. Habibi and S. Tahar, "A Survey on System-On-a-Chip Design Languages," in *Proceedings IEEE International Workshop on System-on-Chip*, Alberta, Canada, June-July 2003.

[3] C. Rowen, "Reducing SoC Simulation and Development Time," *IEEE Computer*, vol. 35, no. 12, pp. 29–34, December 2002.

[4] J. Henkel, "Closing the SoC Design Gap," *IEEE Computer*, vol. 36, no. 9, pp. 119–121, September 2003.

[5] F. Pichon, S. Blanc, and B. Candaele, "Mixed-Signal Modelling in VHDL for System-On-Chip Applications," in *Proceedings European Design and Test Conference*, Paris, France, 6-9 March 1995.

[6] J. Barby, S. Rehan, and M. Elmasry, "AHDL Modelling to Support Top-down Design of Mixed-Signal ASICs," in *Proceedings 7th Annual IEEE International ASIC Conference and Exhibit*, New-York USA, 19-23 September 1994.

[7] K. ODA, L. Prado, and A. Gadient, "A New Methodology for Analog/Mixed-Signal (AMS) SoC Design That Enables AMS Design Reuse

and Achieves Full-Custom Performance," in *Proceedings IEEE/DATC Electronic Design Processes Workshop*, Monterey, USA, 21-23 April 2003.

[8] P. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann, 2000.

[9] IEEE Standard for Verilog Hardware Description Language, *IEEE Std. 1364-2001*, 13th edition, 2001.

[10] Open SystemC Initiative OSCI, *SystemC Language Reference Manual*, www.systemc.org, 2003.

[11] Accellera, *SystemVerilog language Reference Manual 3.1*, 2004.

[12] W. Banzhaf, *Computer-Aided Circuit Analysis using SPICE*, Prentice Hall, 1989.

[13] D. Hanselman and B. Littlefield, *Mastering MATLAB 6*, Prentice Hall, 2003.

[14] IEEE Inc, *IEEE Standard VHDL language Reference Manual (Integrated with VHDL-AMS Changes),IEEE std 1076.1*, 1997.

[15] Open Verilog International, *Verilog-AMS Language Reference Manual 2.0*, January 2000.

[16] R. Roth and D. Ramanathan, "A High-level Hardware Design Methodology Using C/C++," in *Proceedings High Level Design Validation and Test Workshop*, San Diego, USA, 1999.

[17] G. Arnout, "C for System Level Design," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 9-12 March 1999.

[18] S. liao, "Towards a New Standard for System-Level Design," in *Proceedings International Symposium on Hardware/Software Codesign*, San Diego California USA, 3-5 May 2000.

[19] Computing Labortary, *Handel-C Manual*, Oxford University.

[20] D. Gajski and J. Zhu, *SpecC: Specification language and Design Methodology*, Kluwer Academic Publishers, 2000.

[21] Open SystemC Initiative OSCI Documents, *SystemC 2.0.1 User's Guide*, 1996-2002.

[22] D. Leenaerts, G. Gielen, and R. Rutenbar, "CAD Solutions and Outstanding Challenges for Mixed-Signal and RF IC Design," in *Proceedings IEEE/ACM International Conference on Computer Aided Design*, San Jose California USA, 4-8 November 2001.

[23] G. Gielen and R. Rutenbar, "Computer-Aided Design of Analog and Mixed-Signal Integrated Circuits," *IEEE*, vol. 88, no. 12, pp. 1825 – 1854, December 2000.

[24] K. Kundert, H. Chang, D. Jefferies, G. Lamant, E. Malavasi, and F. Sendig, "Design of Mixed-Signal Systems-on-a-Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1561 – 1571, December 2000.

[25] Blue Pacific, *BlueHDL User's Manual*, at www.bluepc.com, version 09-20-01 edition, 2001.

[26] synopsys Products and Solution, *System Studio Datasheet*, at www.synopsys.com, 2003.

[27] Forte Design Systems, *Cynthesizer Datasheet*, at www.forteds.com, 2004.

[28] EDA today, *Electronic Design Automation User Study*, 1999.

[29] Celoxica, *Survey of System Design Trends*, December 2003.

[30] ICCAD 2000 Roundtable, *IEEE Design and Test of Computers*, May-June 2001.

[31] Nikkei Publications NEAsia Online, *C Language in Chip Design Shorten Time-to-Market*, September 2004.

[32] L. Semeria, *Applying Pointer Analysis to the Synthesis of Hardware From C*, Phd thesis, Standford University, June 2001.

[33] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara, "From VHDL Register Transfer Level to SystemC Transaction Level Modeling: a Comparative Case Study," in *Proceedings 16th Symposium on Integrated Circuits and Systems Design*, Sao Paulo, BRAZIL, 8-11 September 2003.

[34] Open SystemC Initiative OSCI Documents, "Overview of the Open SystemC Initiative," 1999.

[35] Mentor Graphics Corporation, *Seamless Datasheet*, at www.mentor.com, 5th edition.

[36] J. Kunkel and K. Kranen, "SystemC Demonstrates Rapid Progress," EE Times, 26 September 2000.

[37] D. Smith, S. Majdecki, and D. Johmson, "Interactive Control of Analog System Simulation," *VLSI Systems Design*, vol. 8, no. 7, pp. 46–54, 1987.

[38] D. Thelen and J. MacDonald, "Simulating Mixed Analog-Digital Circuits on a Digital Simulator," in *Proceedings IEEE International Conference on Computer-Aided Design*, 7-10 November 1988.

[39] D. Metzner and J. Schafer, "Architecture Development of Mixed Signal ICs for Automotive Applications," in *Proceedings IEEE International Behavioral Modeling and Simulation Conference*, Santa Rosa, California, USA, 7-8 October 2002.

[40] T. Zhang, K. Chakrabarty, and R. Fair, "Behavioral Modeling and Performance Evaluation of Microelectrofluidics-Based PCR Systems Using SystemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 6, pp. 843 – 858, June 2004.

[41] S. Levitan, J. Martinez, T. Kurzweg, A. Davare, M. Kahrs, M. Bails, and D. Chiarulli, "System Simulation of Mixed-Signal Multi-Domain Microsystems With Piecewise Linear Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 2, pp. 139–156, February 2003.

[42] D. Edenfeld, A. Kahng, M. Rodgers, and Y. Zorian, "2003 Technology Roadmap for Semiconductors," *IEEE Computer Society*, pp. 47–56, January 2004.

[43] Southampton University, *VHDL-AMS Validation Suite*, http://www.syssim.ecs.soton.ac.uk/validsui.html, 1997.

[44] J. Bhasker, *A SystemC Primer*, Star Galaxy Publishing, 2002.

[45] A. Vachoux, Ch. Grimm, and K. Einwich, "SystemC-AMS Requirements, Design Objectives and Rationale," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[46] Open SystemC Initiative OSCI Documents, *Functional Specification for SystemC 2.0.1, Version 2.0-Q*, 2002.

[47] J. Gerlach and W. Rosenstiel, "System Level Design Using the SystemC Modelling Platform," in *Proceedings Forum on Specification and Design Languages*, Copenhagen Denmark, 26-29 June 2001.

[48] Open SystemC Initiative OSCI Documents, *An Introduction to System Level Modelling in SystemC 2.0*, 2001.

[49] P. Panda, "SystemC- A modelling Platform Supporting Multiple Design Abstractions," in *Proceedings 14th International Symposium on Systems Synthesis*, Montreal Canada, 1-3 October 2001.

[50] H. Patel, "HEMLOCK: Heterogeneous Model of Computation Kernel for SystemC," M.S. thesis, Virginia Polytechnic Institute and State University, December 2003.

[51] S. Pasricha, "Transaction Level Modeling of SoC Using SystemC 2.0," in *Proceedings Synopsys User Group Conference*, Bangalore, India, May 2002.

[52] R. Hilderink and S. Klostermann, "Transaction Level Modelling of SOC Platform Using SystemC," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 4-8 March 2002.

[53] D. Ku and G. De Micheli, *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*, Kluwer Academic Publishers, Boston, USA, 1992.

[54] Ch. Stoud, R. Munoz, and D. Pierce, "Behavioral Model Synthesis with Cones," *IEEE Design and Test of Computers*, vol. 5, no. 3, pp. 22–30, June 1988.

[55] L. Lavagno and E. Sentovich, "ECL: A Specification Environment for System-Level Design," in *Proceedings Design Automation Conference*, New Orleans, USA, June 1999.

[56] N. Koichi, K. Andrew, Y. Akihisa, K. Takashi, and N. Toshio, "Hardware Compiler BachC," Tech. Rep., Sharp Laboratories of Europe Limited, 2001.

[57] IMAC, *OCAPI*, http://www.imec.be/ocapi.

[58] CynApps, *CYNLIB*, http://www.cynapps.com.

[59] M. O'Nils J. Lundgren and B. Oelmann, "A SystemC Extension for Behavioural Level Quantification of Noise Coupling in Mixed-Signal Systems," in *Proceedings IEEE International Symposium on Circuits and Systems*, Bangkok, Thailand, 25-28 May 2003.

[60] T. Bonnerud, B. Hernes, and T. Ytterdal, "A Mixed-Signal Functional Level Simulation Framework Based on SystemC," in *Proceedings IEEE Custom Integrated Circuits Conference*, San Diego California USA, 6-9 May 2001.

[61] M. Conti, M. Caldari, S. Orcioni, and G. Biagetti, "Analog Circuit Modelling in SystemC," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[62] S. Orcioni, G. Biagetti, and M. Conti, "SystemC-WMS: A Wave Mixed Signal simulator," in *Proceedings Forum on Specification and Design Langauges*, Lausanne, Switzerland, 27-30 September 2005.

[63] Ch. Grimm, Ch. Meise, W. Heupke, and K. Waldschmidt, "Refinement of Mixed-signal Systems with SystemC," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[64] J. Romberg and Ch. Grimm, "Refinement of Hybrid Systems From Formal Models to Design Languages," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[65] K. Einwich, Ch. Clauss, G. Noessing, P. Schwarz, and H. Zojer, "SystemC Extensions for Mixed-Signal System Design," in *Proceedings Forum on Specification and Design Languages*, Lyon France, 3-7 September 2001.

[66] K. Einwich, "Application of SystemC/SystemC-AMS for the Specification of a Complex Wired Telecommunication System," in *Proceedings Forum on*

Specification and Design Langauges, Lausanne, Switzerland, 27-30 September 2005.

[67] E. Markert, G. Herrmann, and D. Muller, "System Model of an Inertial Navigation System Using SystemC-AMS," in *Proceedings Forum on Specification and Design Langauges*, Lausanne, Switzerland, 27-30 September 2005.

[68] Ch. Grimm, P. Oehler, Ch. Meise, K. Waldschmidt, and W. Fey, "AnalogSL: A Library for Modelling Analog Power Drivers with C++," in *Proceedings Forum on Specification and Design Languages*, Lyon France, 3-7 September 2001.

[69] P. Birrer and W. Hartong, "Incorporating SystemC in Analog/Mixed-Signal Design Flow," in *Proceedings Forum on Specification and Design Langauges*, Lausanne, Switzerland, 27-30 September 2005.

[70] A. Vachoux, Ch. Grimm, and K. Einwich, "Analog and Mixed Signal Modelling with SystemC-AMS," in *Proceedings IEEE International Symposium on Circuits and Systems*, Bangkok, Thailand, 25-28 May 2003.

[71] K. Einwich and Ch. Grimm, "Mixed Signal Extensions for SystemC," in *Proceedings Forum on Specification and Design Languages*, Marseille, France, 24 - 27 September 2002.

[72] K. Einwich, "SystemC-AMS Steps Towards an Implementation," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[73] T. Cuenin, O. Romain, and P. Garda, "Design and modelling of an i2c bus controller," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[74] University of Cincinnati Electronic Design Automation Research Centre, Distributed Processing Laboratory, *VHDL-AMS Analyzer Ver 0.4*, http://www.ece.uc.edu/ vasu/index.html, 1997.

[75] *VHDL-AMS Frontend, Java- Version*, http://www.ti.informatik.uni-frankfurt.de/grimm/hybrid.html#VHDL-AMS, 1997.

[76] LEDA S.A., *VHDL-AMS compiler*, http://worldserver.oleane.com/leda.

[77] D. Muller, "Subproject A2-System-Level Modelling and System Simulation with VHDL-AMS Examplified by a Full Circle Panoramic," in *Proceedings Microsystem Symposium*, Delft, The Netherlands, 10-11 September 1998.

[78] E. Moser, "VHDL-AMS the Missing Link in System Design Experiment with Unified Modelling in Automotive Engineering," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 23-26 February 1998.

[79] I. Sharp, O. Zinke, and A. Patterson, "Using VHDL-AMS for a Motor-Controller Design and Simulation," in *Proceedings DAK Forum*, Trondheim, Norway, 27-28 October 1999.

[80] J. Hanna and R. Hillman, "A Mixed Basis for Micro-System Modelling," in *Proceedings International Conference on Modelling and Simulation of Microsystems*, San Juan Puerto Rico USA, 19-21 April 1999.

[81] J. Alejandro L. Alcantud and T. Kazmierski, "VHDL-AMS Modelling of Self-Organizing Neural Systems," in *Proceedings IEEE International Symposium on Circuits and Systems*, Geneva, Switzerland, 28-31 May 2000.

[82] N. Milet, G. Monnerie, A. Fakhkakh, D. Geofferoy, Y. Herve, H. LEVI, and J. Charlot, "A VHDL-AMS Library of RF Blocks Models," in *Proceedings IEEE International Workshops on Behavioral Modeling and Simulation*, Santa-Rosa USA, 11-12 October 2001.

[83] C. Lallement, F. Pcheux, and Y. Herv, "VHDL-AMS Design of a MOST Model Including Deep Submicron and Thermal-Electronic Effects," in *Proceedings IEEE International Workshops on Behavioral Modeling and Simulation*, Santa-Rosa USA, 11-12 October 2001.

[84] F. Hamid and T. Kazmierski, "Analogue Filter Synthesis From VHDL-AMS," in *Proceedings Forum on Specification and Design Languages*, Lyon, France, 3-7 September 2001.

[85] A. Doboli, "The Definition of VHDL-AMS Subset for Behavioural Synthesis of Analogue Systems," in *Proceedings IEEE International Workshops on Behavioral Modeling and Simulation*, Orlando USA, 27-28 October 1998.

[86] U. Heinkle, M. Padeffke, W. Haas, T. Buerner, H. Braisz, T. Gentner, and A. Grassmann, *The VHDL Reference: Practical Guide to Computer-Aided Integrated Circuit Design including VHDL-AMS*, Wiley Publisher, 2000.

[87] P. Ashenden, G. Peterson, and D. Teegarden, *The System Designer's Guide to VHDL-AMS*, Morgan Kaufmann, 2000.

[88] E. Christen and K.Bakalar, "VHDL-AMS-A Hardware Description Language for Analogue and Mixed-Signal Applications," *IEEE Trans. On Circuit and Systems-II: Analogue and Digital Signal Processing*, vol. 46, no. 10, pp. 1263–1272, October 1999.

[89] Jesse Liberty, *Teach yourself C++ in 24 hours*, Sams Publishing, 3rd edition, 2002.

[90] C. Ho, A. Ruehli, and P. Brennan, "The Modified Nodal Approach to Network Analysis," *IEEE Transactions on Circuit and systems*, vol. CAS-22, no. 6, June 1975.

[91] K. Nichols, T. Kazmierski, M. Zwolinski, and A. Brown, "Overview of SPICE-Like Circuit Simulation Algorithms," *IEE circuits Devices Systems*, vol. 141, no. 4, pp. 242–250, Augest 1994.

[92] K. Nichols, J. Lin, A. Brown, T. Kazmierski, and M. Zwolinski, "Reliability of circuit-level simulation," in *Proceedings IEE Colloquium on SPICE: Surviving Problems in Circuit Evaluation*, 30 January 1993.

[93] T. Kazemirski, "Fuzzy-Logic Digital-Analogue Interfaces for Accurate Mixed-Signal Simulation," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 23-26 February 1998.

[94] M. Daniel and C. Gwyn, "CAD Systems for IC Design," *IEEE Tansactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-I, no. 1, January 1982.

[95] D. Calahan, *Computer Aided Network Design*, McGraw-Hill, 1972.

[96] L. Chua and P. Lin, *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Prentice Hall, 1975.

[97] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, Van Nostrand Reinhold, 1983.

[98] A. Ruehli, *Circuit Analysis, Simulation and Design*, North-Holland, 1986.

[99] V. Litovski and M. Zwolinski, *VLSI Circuit Simulation and Optimization*, Chapman and Hall, 1997.

[100] T. Kazmierski, *Chapter 3 Techniques of Circuit Simulation in Computer-Aided Tools for VLSI System Design*, Peter Peregrinus Ltd., Stevenage, U.K., 1986.

[101] H. Shichman, "Integration System of a Nonlinear Network Analysis Program," *IEEE Transactions on Circuit Theory*, vol. CT-17, pp. 378–386, August 1970.

[102] W. Van Bokhoven, "Linear Implicit Differentiation Formulas of Variable Step and Order," *IEEE Transactions on Circuits and Systems*, vol. 22, no. 2, pp. 109–115, February 1975.

[103] E. Haber, "Quasi-Newton Methods for Large-Scale Electromagnetic Inverse Problem," *IOP publishing inverse problems*, , no. 21, pp. 305–323, 2005.

[104] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, Th. Kropf, and W. Rosenstiehl, "The Simulation Semantics of SystemC," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 13-16 March 2001.

[105] T. Kujanpaa, "Mixed Analog/Digital Circuit Simulation in APLAC," Tech. Rep. CT-37, Helsinki University of Technology, Department of Electrical and Communication Engineering, Circuit Theory laboratory, 1998.

[106] D. Lungeanu and C. Richard Shi, "Distributed Event-Driven Simulation of VHDL-SPICE Mixed-Signal Circuits," in *Proceedings International Conference on Computer Design*, Texas, USA, 23-26 September 2001.

[107] D. Overhauser and R. Saleh, "Evaluating Mixed-Signal Simulators," in *Proceedings IEEE Custom Integrated Circuits Conference*, Santa Clara CA, USA, May 1995.

[108] M. Zwolinski, C. Garagate, Z. Mrcarica, T. Kazmierski, and A. Brown, "Anatomy of a Simulation Backplane," *IEE Computers and Digital Techniques*, vol. 142, no. 6, pp. 377 – 385, November 1995.

[109] Analogy Inc, *Guide to Mixed-Signal Simulation, Book one: VHDL-AMS*, 1996-1999.

[110] A. Brown and M. Zwolinski, "The Continuous-Discrete Interface - What Does This Really Mean? Modelling and Simulation Issues ," in *Proceedings International Symposium on Circuits and Systems*, 25-28 May 2003.

[111] S. Strogatz, *Nonlinear Dynamics and Chaos*, Addison-Wesley, 1994.

[112] MathWork, *MATLAB, Vander Pol Oscillator demo (deedemo1).*

[113] Mentor Graphics Corporation, www.mentor.com, *System Vision Data sheet.*

[114] J. Gleick, *Chaos: Making a New Science*, Minerva Publisher, November 1996.

[115] MathWork, *MATLAB, Lorenz Chaos demo (deedemo2).*

[116] S. Ben-Yaakov, "SPICE Simulation of PWM DC-DC Convertor Systems: Voltage Feedback, Continuous Inductor Conduction Mode," *IEE Electronics Letters*, vol. 25, no. 16, pp. 1061 – 1063, August 1989.

[117] B. Antao, F. El-Turky, and R. Leonowich, "Mixed-Mode Simulation of Phase-Locked Loops," in *Proceedings IEEE Custom Integrated Circuits Conference*, 9-12 May 1993.

[118] N. Godambe and C. Richard Shi, "Behavioral Level Noise Modeling and Jitter Simulation of Phase-Locked Loops With Faults Using VHDL-AMS," in *Proceedings IEEE VLSI Test Symposium*, 27 April-1 May 1997.

[119] M. Takahashi, K. Ogawa, and K. Kundert, "VCO Jitter Simulation and its Comparison with Measurement," in *Proceedings Asia and South Pacific Design Automation Conference*, Wanchai, Hong Kong, 18-21 January 1999.

[120] X. Mao, H. Yang, and H. Wang, "Behavioral Modeling and Simulation of Jitter and Phase Noise in Fractional-N PLL Frequency Synthesizer," in *Proceedings IEEE International Behavioral Modeling and Simulation Workshop*, San Jose, California, USA, 21-22 October 2004.

[121] D. Jiles and D. Atherton, "Theory of Ferromagnetic Hysteresis," *Journal of Applied Physics*, vol. 55, no. 6, pp. 2115–2120, 15 March 1984.

[122] D. Jiles and D. Atherton, "Theory of Ferromagnetic Hysteresis," *Journal of Magnetism and Magnetic Materials*, vol. 61, no. 1-2, pp. 48–60, September 1986.

[123] D. Jiles and J. Thoelke, "Theory of Ferromagnetic Hysteresis: Determination of Model Parameters from Experimental Hyteresis Loops," *IEEE Transactions on Magnetics*, vol. 25, no. 5, pp. 3928–3930, September 1989.

[124] C. Chuang and C. Harrison, "Analogue Behavioural Modelling and Simulation Using VHDL and Saber-MAST," in *Proceedings IEE Colloquium on Mixed Mode Modelling and Simulation*, November 1994.

[125] A. Maxim, D. Andreu, and J. Boucher, "A Novel Behavioral Method of SPICE Macromodeling of Magnetic Components Including The Temperature and Frequency Dependencies," in *Proceedings IEEE Applied Power Electronics Conference and Exposition*, USA, 15-19 February 1998.

[126] P. Wilson and T. Kazmierski, "A Novel Approach to Mixed-Domain Behavioral Modeling of Ferromagnetic Hysteresis in VHDL-AMS," in *Proceedings Forum on Specification and Design Langauges*, Lille-France, 13-17 September 2004.

[127] P. Wilson, J. Ross, A. Brown, T. Kazmierski, and J. Baranowski, "Efficient Mixed-Domain Behavioural Modelling of Ferromagnetic Hysteresis implemented in VHDL-AMS," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 16-20 February 2004.

[128] T. Kazmierski and J. Baranowski, "A Modified Jiles-Atherton Model of Ferromagnetic Hysteresis for Behavioral Circuit Simulation in VHDL-AMS,"

in *Proceedings IEEE Behavioral Modeling and simulation Workshop*, Orlando,USA, October 1999.

[129] M. Williams, R. Vogelsong, and K. Kundert, "Simulation and Modeling of Nonlinear Magnetics," in *The Designer Guide*, www.designers-guide.com, March 2002.

[130] W. Archer, M. Deveney, and R. Nagel, "Non-Linear Transformer Modeling and Simulation," in *Proceedings IEEE Midwest Symposium on Circuits and Systems*, LA USA, 3-5 August 1994.

[131] H. Brachtendorf and R. Laur, "A Hysteresis Model for Hard Magnetic Core Materials," *IEEE Transactions on Magnetics*, vol. 33, no. 1, pp. 723 – 727, January 1997.

[132] K. Ngo, "Subcircuit Modeling of Magnetic Cores with Hysteresis in PSpice," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 38, no. 4, pp. 1425–1434, October 2002.

[133] E. Della Torre and F. Vajda, "Properties of Accommodation Models," *IEEE Transactions on Magnetics*, vol. 31, no. 3, pp. 1775–1780, May 1995.

[134] A. Brown, J. Ross, and K. Nichols, "Time-Domain Simulation of Mixed Nonlinear Magnetic and Electronic Systems," *IEEE Transactions on Magnetics*, vol. 37, no. 1, pp. 522–532, January 2001.

[135] K. Carpenter, "A Differential Equation Approach to Minor Loops in the Jiles-Atherton Hysteresis Model," *IEEE Transactions on Magnetics*, vol. 27, no. 6, pp. 4404–4406, November 1991.

[136] A. Benabou, S. Clnet, and F. Piriou, "Comparison of Preisach and JilesAtherton Models to Take into Account Hysteresis Phenomenon for Finite Element Analysis," *Journal of Magnetism and Magnetic Materials*, vol. 261, no. 1-2, pp. 139–160, April 2003.

[137] F. Ossart and G. Meunier, "Comparision Between Various Hysteresis Models and Experimental Data," *IEEE Transactions on Magnetics*, vol. 26, no. 5, pp. 2837–2839, September 1990.

[138] F. Preisach, "Uber Die Magnetische Nachwirkung," *Zeitschrift Fur Physik*, p. 277302, 1935.

[139] X. Liu and J. Wagner, "Design of Vibration Isolation Actuator for Automotive Seating Systems-Part1:Modelling and passive isolator performance," *Journal of Vehicle Design*, vol. 29, no. 4, pp. 335–356, April 2002.

[140] X. Liu and J. Wagner, "Design of Vibration Isolation Actuator for Automotive Seating Systems-Part2:Controller design and actuator performance," *Journal of Vehicle Design*, vol. 29, no. 4, pp. 357–375, April 2002.

[141] L. Wang and T.J. Kazmierski, "VHDL-AMS Modeling of an Automotive Vibration Isolation Seating System," in *Proceedings 3rd International Conference on Circuits, Signals and Systems*, CA USA, 24-26 October 2005.

[142] H. Schlebusch, "SystemC Based Hardware Synthesis Becomes Reality," in *Proceedings 26th EUROMICRO Conference*, Maastricht, The Netherlands, 5-7 September 2000.

[143] P. Cavalloro, A. Allara, M. Bombana, and F. Ferrandi, "Requirements for Synthesis Oriented Modeling in SystemC," in *Proceedings Forum on Specification and Design Languages*, Lyon France, 3-7 September 2001.

[144] H. Jang, M. Kang, M. Lee, K. Chae, K. Lee, and K. shim, "High-Level System Modelling and Architecture Exploration with SystemC on a Network SoC: S3C2510 Case Study," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 16-20 February 2004.

[145] A. Braun, T. Schubert, M. Stark, K. Haug, J. Gerlach, and W. Rosenstiel, "Case Study: SystemC-Based Design of an Industrial Exposure Control Unit," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[146] Cadence Design Systems homepage, http://www.cadence.com.

[147] T. Kogel, A. Wieferink, H. Meyr, and A. Kroll, "SystemC Based Architecture Exploration of 3D Graphics Processor," in *Proceedings IEEE Workshop on Signal Processing Systems*, Antwerp, Belgium, 26-28 September 2001.

[148] Coware Inc., *Success Stories*, http://www.coware.com/.

[149] M. Caldari, M. Conti, P. Crippa, G. Marozzi, F. Di Gennaro, and C. Turchetti, "SystemC Modelling of a Bluetooth Transeiver: Dynamic Management of Packet Type in a Noisy Channel," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[150] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralist, and C. Turchetti, "Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[151] F. Bruschi, F. Ferrandi, D. Sciuto, and M. Bombana, "SystemC Specification of a Telecom PCI-Compatible Interface," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 4-8 March 2002.

[152] J. Armstrong and Y. Ronen, "Modelling with SystemC: A Case Study," in *Proceedings Conference on Hardware Description Languages*, San Jose, USA, 11-12 March 2002.

[153] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "SystemC Cosimulation and Emulation of Multiprocessor SoC Designs," *IEEE Computer*, vol. 36, no. 4, pp. 53–59, April 2003.

[154] P. Gerin, S. Yoo, G. Nicolescu, and A. Jerraya, "Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures," in *Proceedings Asia and South Pacific Design Automation Conference*, Yokohama Japan, 30 January - 2 February 2001.

[155] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systematic Embedded Software Generation from SystemC," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[156] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. Aboulhamid, and F. Boyer, "SPACE: A Hardware/Software SystemC Modelling Platform Including an RTOS," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[157] F. Fummi, S. Martini, G. Perbellini, and M. Poncino, "Native ISS-SystemC Integration of Multi-Processor SoC," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 16-20 February 2004.

[158] N. Agliada, A. Fin, F. Fummi, M. Martignano, and G. Pravadelli, "On the Reuse of VHDL Modules into SystemC Designs," in *Proceedings Forum on Specification and Design Languages*, Lyon France, 3-7 September 2001.

[159] L. Mahmoudi, A. Abutalebi, O. Nadjarbashi, and S. Hessabi, "Verilog2SC: A Methodology for Converting Verilog HDL to SystemC," in *Proceedings Conference on Hardware Description Languages*, San Jose, USA, 11-12 March 2002.

[160] M. Bombana and F. Bruschi, "SystemC-VHDL Co-simulation and Synthesis in the HW Domain," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[161] Y. Yuyama, K. Takai, K. Kobayashi, and H. Onodera, "Hardware and Software Codesign with Using SystemC and Bach," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 4-8 March 2002.

[162] L. Cai, P. Kritzinger, M. Olivares, and D. Gajski, "Top-Down System Level Design Methodology Using SpecC, VCC and SystemC," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 4-8 March 2002.

[163] Cadence Inc, *Virtual Component Co-design VCC 2.1 Production Documentation*, 1998.

[164] CoWare, *ConvergenSC Product Family Document*, at www.coware.com.

[165] Mentor Graphics Corporation, *ModelSim SE Datasheet*, at www.model.com, version 6.0 edition, 2004.

[166] A. Habibi and S. Tahar, "A Survey: System-on-a-Chip Design and Verification," Tech. Rep., Concordia University, Canada, January 2003.

[167] A. Fin, F. Fummi, M. Martignano, and M. Signoretto, "SystemC: A Homogenous Environment to Test Embedded Systems," in *Proceedings International Symposium on Hardware/Software Codesign*, Copenhagen, Denmark, 25-27 April 2001.

[168] C. Kern and M. Greenstreet, "Formal Verification in Hardware Design: A Survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, pp. 123–193, April 1999.

[169] D. Grobe and R. Drechsler, "Formal Verification of LTL Formulas for SystemC Designs," in *International Symposium on Circuits and Systems*, Bangkok, Thailand, 25-28 May 2003.

[170] C. Norris and S. Swan, "Using Transaction-Based Verification in SystemC," Tech. Rep., Cadence Design Systems, www.cadence.com, June 2002.

[171] S. Cox, M. Glasser, W. Grundmann, C. Norris, W. Paulsen, J. Pierce, J. Rose, D. Shea, and K. Whiting, "Creating a C++ Library for Transaction-Based Test Bench Authoring," in *Proceedings Forum on Specification and Design Languages*, Lyon France, 3-7 September 2001.

[172] J. DeGroat, A. Raman, and B. Younis, "A Design Project for System Design with SystemC," in *Proceedings IEEE International Conference on Microelectronic Systems Education*, California USA, 1-2 June 2003.

[173] R. Jindal and K. Jain, "Verification of Transactional-Level SystemC Models Using RTL Testbenches," in *Proceedings ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, Mont Saint-Michel France, 24 - 26 June 2003.

[174] L. Semeria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++," in *Proceedings IEEE International High Level Design Validation and Test Workshop*, San Diego USA, November 1999.

[175] A. Hoffmann, T. Kogel, and H. Meyr, "A Framework for Hardware/Software Co-verification," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 13-16 March 2001.

[176] G. Post, P. Venkataraghavan, T. Ray, and D. Seetharaman, "A SystemC-Based Verification Methodology for Complex Wireless Software IP," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 16-20 February 2004.

[177] A. Fin, F. Fummi, and G. Pravadelli, "AMLETO: A Multi-Language Environment for Functional Test Generation," in *Proceedings International Test Conference*, Maryland USA, November 2001.

[178] A. Fin and F. Fummi, "LAERTE++: An Object Oriented High-level TPG for SystemC Designs," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[179] F. Bruschi, F. Ferrandi, M. Chiamenti, D. Sciuto, and P. Milano, "Error Simulation Based on the SystemC Design Description Language," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 4-8 March 2002.

[180] F. Ferrandi, M. Rendine, and D. Sciuto, "Functional Verification for SystemC Descriptions Using Constraint Solving," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 4-8 March 2002.

[181] R. Ramaswamy and R. Tessier, "The Integration of SystemC and Hardware-Assisted Verification," in *Proceedings International Conference on Field-Programmble Logic and Applications*, Montpelier, France, September 2002.

[182] Members of the SystemC Verification Working Group, "SystemC Verification Standard Specification," 2002.

[183] F. Carbognani, Ch. Lennard, C. Norris, A. Cochrane, and P. Bates, "Qualifying Precision of Abstract SystemC Models Using the SystemC Verification Standard," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[184] CoWare, *CoWare N2C SystemDesigner Datasheet*, at www.coware.com, 2003.

[185] Celoxica, *Nexus-PDK Datasheet*, at www.Celoxica.com, 2003.

[186] Summit Design, *Elite Datasheet*, at www.summit-design.com, 2004.

[187] E. Grimpe and F. Oppenheimer, "Extending the SystemC Synthesis Subset by Object-Oriented Features," in *Proceedings Hardware/software Codesign and System Synthesis*, California USA, 1-3 October 2003.

[188] G. De Micheli, "Hardware Synthesis From C/C++ Models," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 9-12 March 1999.

[189] E. Grimpe and F. Oppenheimer, "Aspects of Object-Oriented Hardware Modeling with SystemC-Plus," in *Proceedings Forum on Specification and Design Languages*, Lyon France, 3-7 September 2001.

[190] G. Economakos, P. Oikonomakos, and I. Panagopoulos, "Behavioural Synthesis with SystemC," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 13-16 March 2001.

[191] E. Bernard and R. Mueller, "SystemC: A Case Study on Behavioural Synthesis and Simulation Performance," in *Proceedings Forum on Specification and Design Languages*, Lyon France, 3-7 September 2001.

[192] F. Bruschi and F. Ferrandi, "Synthesis of Complex Control Structures From Behavioural SystemC Models," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[193] L. Charest, M. Reid, E. Aboulhamid, and G. Bois, "A Methodology for Interfacing Open Source SystemC with a Third Party Software," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 13-16 March 2001.

[194] R. Moigne, O. Pasqier, and J-P. Calvez, "A Graphical Tool for System-Level Modeling and Simulation with SystemC," in *Proceedings Forum on*

Specification and Design Languages, Frankfurt, Germany, 23-26 September 2003.

[195] D. Grobe, R. Dorechsler, L. Linhard, and G. Angst, "Efficient Automatic Visualization of SystemC Designs," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[196] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia, "IPSIM: SystemC 3.0 Enhancements for Communication Refinement," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 3-7 March 2003.

[197] R. Siegmund and D. Muller, "SystemCSV: An Extension of SystemC for Mixed Multi-Level Communication Modelling and Interface-Based System Design," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Messe Munich, Germany, 13-16 March 2001.

[198] F. Herrera, P. Sanchez, and E. Villar, "Modelling of CSP, KPN and SR Systems with SystemC," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.

[199] D. Perez, G. Mouchard, and O. Temam, "A New Optimized Implementation of the SystemC Engine Using Acyclic Scheduling," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 16-20 February 2004.

[200] D. Grobe, G. Fey, and R. Drechsler, "Modeling Multi-Valued Circuits in SystemC," in *Proceedings 33rd International Symposium on Multiple Valued Logic*, Tokyo, Japan, 16-19 May 2003.

[201] H. Posadas, F. Herrera, P. Sanchez, E. Villar, and F. Blasco, "System-Level Performance Analysis in SystemC," in *Proceedings Design, Automation and*

Test in Europe Conference and Exhibition, Paris, France, 16-20 February 2004.

[202] O. Blaurock, "C-Model Integration and Software Development Using System-Level Simulation at TLM in a SystemC-based Design Flow," in *Proceedings Forum on Specification and Design Languages*, Frankfurt, Germany, 23-26 September 2003.