TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

An Open Tool Integration Environment for Efficient Design of Embedded Systems in Wireless Communications

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

eingereicht an der
Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Pavle Belanović B.E., M.S
Dietrichsteingasse 5/15, 1090 Wien
geboren in Belgrad am 5. September 1976
Matrikelnummer 0227318

Wien, im Februar 2006 .....

Advisor

Univ.Prof. Dipl.-Ing. Dr.techn. Markus Rupp
Technische Universität Wien
Institut für Nachrichtentechnik und Hochfrequenztechnik

Examiner

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Technische Universität Wien
Institut für Technische Informatik

ABSTRACT

The design of embedded computer systems for modern wireless communication devices finds itself under increasing technological and commercial pressures. This design crisis is fueled by an unrelenting growth in algorithmic complexity, which by far outpaces the growth in design productivity, thus making it increasingly difficult to design entire embedded systems. On the other hand, the commercial reality in the wireless communications sector dictates ever shortening design cycles to achieve quicker time to market.

This thesis examines the traditional design process of embedded systems for wireless communications, identifies the key bottlenecks which inhibit increased design productivity, and proposes the Open Tool Integration Environment (OTIE) as an effective means of removing these bottlenecks. A flexible, scalable, robust, and secure implementation of OTIE is presented, based on a Single System Description (SSD), providing a single, central repository for all refinement information in the design process. The presented implementation also includes proof-of-concept implementations of System Description Interfaces (SDIs), visualizers, as well as two complete and fully functional tool chains.

The first of the tool chains is used for virtual prototyping and demonstrates the ability of OTIE to include both commercial and academic Electronic Design Automation (EDA) tools seamlessly into a complete design flow. The ability of OTIE to increase design productivity through this increased automation is demonstrated, resulting in savings in the order of thousands of person-hours. Also, the virtual prototyping tool chain demonstrates the ability of OTIE to automate both the design and verification flows. The second tool flow, *fixify*, is a toolset dedicated to automating the floating-point to fixed-point conversion, a design step which has not been automated previously. The incorporation of *fixify* demonstrates the ability of OTIE to support new and emerging EDA tools as they become available in the future.

ZUSAMMENFASSUNG

Die Entwicklung von eingebetteten Systemen für moderne drahtlose Kommunikationssysteme steht unter immer stärker werdenden technologischem und hohen kommerziellem Druck. Weiters wird die so genannte Entwurfskrise (Design Crisis) durch das andauernde Wachstum der algorithmischen Komplexität verschärft, die die Steigerung der Produktivität bei weitem übertrifft. Zusätzlich erfordern die wirtschaftlichen Rahmenbedingungen im Mobilkommunikationsbereich immer kürzer werdende Entwicklungszeiten, um ein Produkt möglichst früh am Markt zu platzieren.

In dieser Arbeit wird der traditionelle Entwicklungsprozess für eingebettete Mobilkommunikationssysteme betrachtet und jene Probleme identifiziert, welche eine erhöhte Produktivität verhindern. Es wird eine offene Entwicklungsumgebung (Open Tool Integration Environment - OTIE) als geeignetes Mittel zur Behebung der Entwicklungsprobleme vorgestellt. Die flexible, erweiterbare, robuste und sichere Implementierung von OTIE basiert auf einer Single System Description (SSD), welche eine zentrale Datenbasis für die gesamten Informationen des Entwicklungsprozesses bietet. Die Leistungsfähigkeit dieses Konzepts wird mit der Implementierung der Anbindungen von verschiedenen Systembeschreibungen, einer grafischen Benutzerschnittstelle und zwei automatisierten Entwicklungsabläufen nachgewiesen.

Der erste Entwicklungsablauf integriert die automatische Erzeugung von virtuellen Prototypen (Virtual Prototypes - VPs). Der hohe Automatisierungsgrad der Erzeugung von VPs und des Verifikationsprozesses ermöglicht hierbei Einsparungen in der Größenordnung von mehreren tausend Arbeitsstunden.

Ein herausfordernder Schritt in der Entwicklung von eingebetteten Systemen ist die Konvertierung der Gleitkomma- zur Fixpunktdarstellung für deren optimale Umsetzung bisher noch keine praktikable Lösung existierte. Im zweiten Entwicklungsablauf wird mit dem Werkzeug *fixify* eine automatisierte Lösung dieses Optimierungsproblems erreicht.

An Hand der Integration beider Entwicklungsabläufe in OTIE wird die Möglichkeit gezeigt sowohl kommerzielle als auch akademische Werkzeuge nahtlos in OTIE einzubinden.

ACKNOWLEDGEMENTS

As the author of this thesis, I am keenly aware that it represents the fruition of not only my own work, but also the support which other individuals and organizations have lent me over the years, and for which I am profoundly grateful.

Firstly, I would like to thank Dr. Markus Rupp for giving me the opportunity to engage in this research, for the constant readiness with which he shared his considerable insight and experience with me, and for his inspired guidance throughout the course of my work. I consider myself highly fortunate to have been his student. I would also like to thank the Christian Doppler Research Association and Infineon Technologies for the financial support which made the work described in this thesis possible. My thanks also goes to Dr. Andreas Steininger for his objective, thorough, and extremely constructive feedback on this thesis.

Special recognition is here due to my colleagues Martin Holzer and Bastian Knerr. I have benefited greatly from their rare ingenuity and true friendship, and for this I am deeply grateful.

I would like to express the greatest of gratitude to my parents, Dragan and Zora, as well as my sister Ana, for the extraordinary way they provide me with unfaltering support, encouragement, and love, even when I am half the world away from them. Finally, I offer special thanks to Victoria, for her love, understanding, and inexhaustible kindness.

CONTENTS

1. INTRODUCTION

One of the most pronounced effects technology has had on the human society in the last 100 years has been the rapid and unrelenting growth in the presence of electronic systems in almost every aspect of modern life. In this time span, electronics have gone from inception to omnipresence, pervading the household, the workplace, and the communal infrastructure.

The later half of the 20th century witnessed another revolution: the explosive infusion of computing systems into every pore of the society. Especially prevalent are embedded systems, which are built for an explicit purpose (such as scanning items on a supermarket checkout, or controlling the traction of each wheel of a vehicle) and do not allow for programmability by the end user. Such systems are by definition shrouded from view of the end user and often function completely unobserved. For example, a modern vehicle contains more than 80 embedded systems [1], the presence of most of which is not immediately apparent to the user.

The proliferation of embedded systems has taken place across all commercial sectors, including communications, transportation, energy, education, security, entertainment, and others. Perhaps the most ubiquitous and rapidly growing, the communications sector comprises mass media such as television, radio, and the Internet, as well as personal communications such as wireline and wireless telephone networks.

Particularly strong growth has been seen in the wireless communications domain, where

the global number of mobile subscribers is predicted to reach 1,6 billion by the end of 2005 [2]. This growth in demand has correspondingly created unprecedented pressure on equipment manufacturers and service providers alike. The pressure on service providers resulted in a highly fragmented geo-political map of dozens of incompatible wireless standards. At the same time, the pressure on manufacturers resulted in an extremely strong growth in complexity of user equipment, where the next generation mobile devices for 3G UMTS systems are expected to be based on processors containing more than 40 million transistors [3]. Even more importantly, the commercially viable time-to-market for mobile devices is rapidly shrinking under the competitive market pressure, putting increasing strain on design processes for these devices. It will be shown in Chapter 2 that the current design processes are failing to keep up with this growing algorithmic demand in the wireless domain and hence require urgent enhancements.

It is the objective of the work presented in this thesis to identify weaknesses of the current design processes, propose a new design methodology to eliminate these weaknesses, present an implementation of the methodology, and show its application on realistic design flows as a proof of concept.

The contents of this thesis are divided into five chapters, the first of which contains this introduction. In Chapter 2, the acute inability of the modern design processes to keep up with the rapidly growing complexity of application algorithms is shown, the underlying inefficiencies which cause this growing gap are exposed, and thus the outline of the necessary solution to this problem is drawn. A survey of related work identifies several industrial and academic research initiatives aimed at providing a suitable solution. However, these research efforts are either aimed only at particular refinement levels of the design process (e.g. the SPIRIT consortium [4]), or particular parts of the embedded system (e.g. Model Integrated Computing [5]).

Hence, this thesis proposes an original contribution in the form of the Open Tool Integration Environment (OTIE), providing the first complete solution for all parts of the system and across all refinement levels of the design process [6]. The structure of the proposed environment, as well as the details of its implementation are also discussed.

In Chapter 3, the first tool chain integrated into the OTIE is presented, automating the refinement of the system model from the algorithmic to the Virtual Prototype (VP) level, in both the design and verification flows [7, 8]. The existing body of work in the field of virtual prototyping is surveyed, showing much effort in the use of VPs to speed up the design process. However, very limited amount of research effort has so far gone into achieving the additional efficiency gains of automatic generation of VPs as well.

The virtual prototyping environment presented in this thesis is the first environment to offer fully automated generation of heterogeneous VPs directly from algorithmic descriptions, including support for Dynamic Data Flow (DDF) systems. Also, the environment includes tools for automated refinement of test patterns form the algorithmic to the VP level. As such, the environment is shown to reliably offer savings of hundreds to thousands of person-hours. Even more importantly, the presented environment serves as a proof of concept for the OTIE, showing its ability to seamlessly integrate both commercial and academic tools into complete and efficient tool chains.

A further tool chain is presented in Chapter 4, automating the conversion from floating-point to fixed-point formats. This critical step in the design process is traditionally performed manually, requiring relatively large design effort investments, and is currently not supported by commercial tools. However, relatively recently several academic initiatives for tackling this conversion have made their appearance, each falling into either the static or the dynamic category.

This thesis presents *fixify*, a novel design environment integrated into the OTIE and

offering fully automated floating-point to fixed-point conversion, thus completely disburdening the designer in this critical design step [9, 10]. A thorough analysis of performance of the three different optimization algorithms available in the *fixify* environment is given, along with a discussion on the applicability of each algorithm under various development scenarios.

Finally, in Chapter 5, the conclusions of this thesis are drawn. In addition to the summary of the presented unique contributions, a discussion of the possible future directions of research based on this thesis are presented.

2. OPEN TOOL INTEGRATION ENVIRONMENT

The strong demand on the wireless communications market creates, as already stated, high pressures on service providers and mobile terminal manufacturers alike. This in turn creates a highly innovative market, where increasing competition drives the trend towards ever shorter development cycles and makes faster introduction of products to the market a critical exercise [11]. Because of these commercial pressures, the development processes for wireless embedded systems face a set of unique challenges.

2.1 Problem Statement

2.1.1 Wireless Standards

The unprecedented speed with which wireless communications systems have developed has brought about a rapid succession of standards, each offering improved performance over its predecessors. Figure 2.1 illustrates the progression of wireless standards from their inception (circa 1980) until today.

First generation (1G) wireless communication standards were designed to transmit analogue voice information only. The strong market growth of 30% - 50% per annum [12] at this early stage of development drove rapid and uncoordinated establishment of mobile networks in various regions of the world, resulting in a variety of non-compatible standards (see Figure 2.1).

A similar trend of global fragmentation of standards followed in the second generation

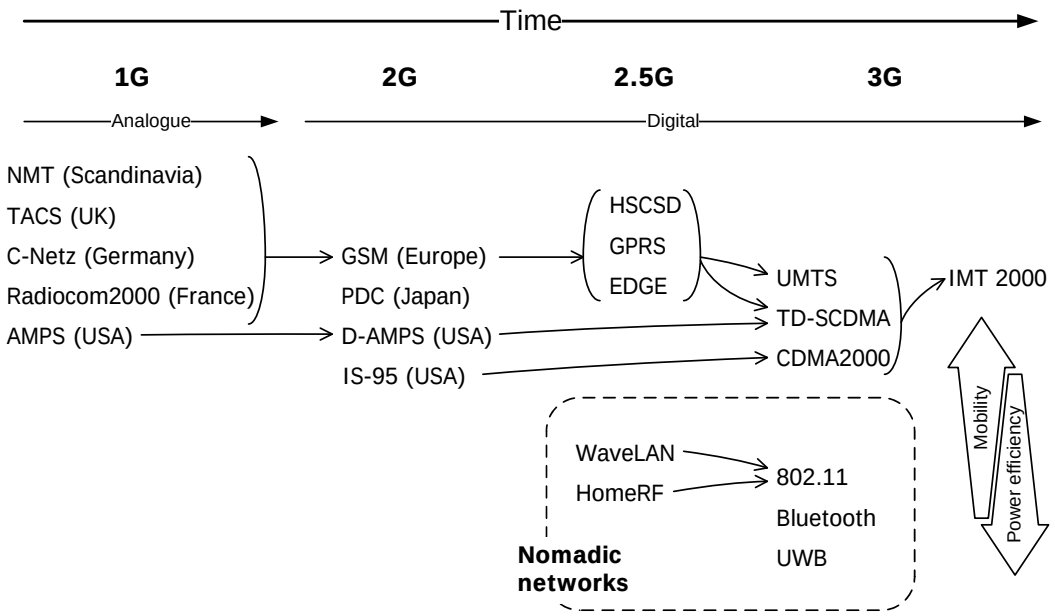

Figure 2.1: *Wireless Standards*

(2G) systems, though the European GSM standard won far wider support than any of its competitors, in 2005 holding a 74% share of the global wireless communications market [13]. Starting with the second generation, mobile networks were designed for transmission of digital data in Time Division Multiple Access (TDMA) and Code Division Multiple Access (CDMA) schemes. Extension of the GSM system into packet data transmission heralded the so-called 2.5G generation of standards, including the GPRS, HSCSD, and EDGE protocols, of which GPRS is the most common.

The current landscape of mobile standards worldwide is marked by extensive deployment of 2G systems, which have reached full maturity, as well as gradual rollout of 3G infrastructure, including UMTS networks in Europe and Asia and CDMA2000 networks in North America. Coexisting in parallel with these 3G networks, a number of shorter range wireless communication standards, such as UWB, Bluetooth, and IEEE 802.11, are helping establish the full spectrum of wireless coverage needed by mobile users for both voice and data communications. In general, these shorter range standards offer (in

Value	CAGR
Demanded algorithmic complexity	74%
Available silicon complexity	59%
Design productivity	21%

Table 2.1: *Comparison of complexity trends*

varying degrees) higher data rates, lower infrastructure cost, and higher power efficiency than 3G systems, but with severely reduced user mobility, both in terms of range and velocity.

2.1.2 Algorithmic and Design Gaps

As mentioned earlier, the evolution of wireless communications standards over their three generations has taken place in only 25 years. During this relatively short period of time, a staggering increase in complexity of over six orders of magnitude has taken place - an estimated 10^6 difference in complexity exists between original 1G systems and current 3G systems [14].

In comparison to this extremely fast-paced growth in algorithmic complexity, the concurrent increase in the complexity of silicon integrated circuits proceeds according to the well-known Moore's Law [15], famously predicting the doubling of the number of transistors integrated onto a single integrated circuit every 18 months.

Lastly, the International Technology Roadmap for Semiconductors [16] reported a growth in design productivity, expressed in terms of designed transistors per staff-month, of approximately 21% Compounded Annual Growth Rate (CAGR).

Bringing all the three growth rates discussed above to the CAGR as a common measure results in the figures shown in Table 2.1. See Appendix B for a detailed discussion of these CAGR calculations.

Hence, it can be concluded that the growth in silicon complexity lags behind the extreme growth in the algorithmic complexity of wireless communication systems. This is known as the **algorithmic gap**. Thus, the abilities of underlying silicon platforms on which wireless communication systems are built have to be exploited with increasing efficiency, i.e. more functionality has to be gained from each individual transistor. In other words, the **quality** of the design process, i.e. effective functionality per unit raw silicon achieved through both hardware and software parts of the system, needs to increase.

It can also be concluded from the data in Table 2.1 that the growth in design productivity lags behind the growth in silicon complexity. This is known as the **design gap**. From this it follows that it is increasingly difficult to design entire integrated circuits - although ever more transistors can be designed over some period of time, over the same period of time the total number of transistors in the circuit increases by an even higher factor. Hence, assuming a constant time to market and barring a continuous increase in team size, the **speed** of the design process, i.e. the ability of the designer to convert specified functionality into circuit layout, has to increase significantly.

Therefore, the existence of the design gap points to inefficiencies in the design process. At various stages in the process, these inefficiencies form bottlenecks, impeding increased productivity, which is needed to keep up with the mentioned algorithmic demand. In order to clearly identify these bottlenecks in the design process, they are classified into **internal** and **external** barriers. The following sections describe the structure of the design process followed by discussions of internal and external barriers to increased productivity.

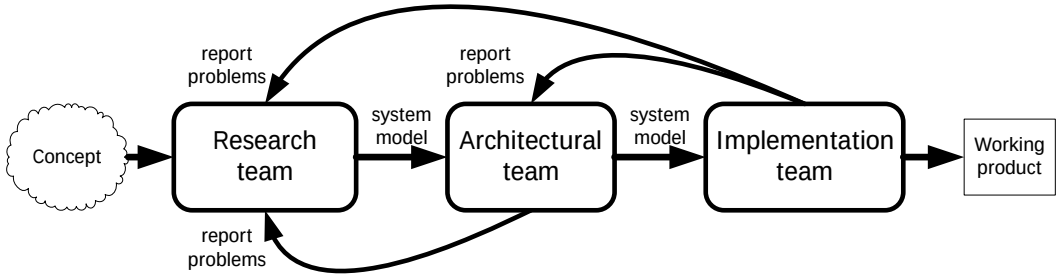

Figure 2.2: *Traditional team structure*

2.1.3 Structure of the Design Process

The flow of the overall design process, starting at the initial conceptual idea of the system and finishing in the final product, is traditionally divided into a number of **abstraction levels**. Of the many design methodologies in existence, each prescribes a different set of abstraction levels to make up the design process, usually tailored to a particular application domain (such as wireless/wireline communications, multimedia processing, automotive, etc.) or the product range of a particular Electronic Design Automation (EDA) tool vendor. As a result, there is no clear and universally accepted division of the design process into a well-defined set of abstraction levels. Rather, there exists a great number of overlapping or even synonymous definitions of abstraction levels, some of which are broad in scope, while others cover small and very specific parts of the design flow, and again some of which enjoy wide recognition in academic and industrial circles, while others are referred to less commonly. Table 2.2 shows a collection of some of the most commonly used abstraction levels in academic literature and/or industrial practice, given in their relative order within the overall design process (from high to low level of abstraction).

In this work we assume the structure of the design process as presented by Rupp et al. [17], consisting of the **research** (or algorithmic), the **architectural**, and the **implementation** teams. This structure is shown in Figure 2.2.

Abstraction Levels
Mission level
System level
Electronic system level
Algorithmic level
Behaviour level
Transaction level
Untimed functional level
Virtual prototype level
Timed functional level
Architecture level
Register-transfer level
Implementation level
Gate level
Transistor level
Wire level

Table 2.2: *Some frequently encountered abstraction levels*

The research team works on new technological ideas and creates the algorithmic model of the new system, presenting purely its functionality without any implementation details. At this level of the design process, focus is placed on modeling a maximally broad view of the system, while keeping the depth of detail with which the system is modeled low, in order to achieve reasonable simulation times. Hence, algorithmic system descriptions generally contain no timing information, system components are described in implementation-general form (i.e. they are not yet assigned for implementation in hardware or software), and simulation results are not bit-precise. However, these models do simultaneously cover all parts of the system, investigate their interactions, offer insight into communication bottlenecks, as well as relative computational complexity of individual parts of the system.

The architectural team concerns itself with the building blocks that will be needed to implement the new system, and specifies the required architecture, including both hardware and software components. Hence, the focus of model descriptions at this level is the structure of the system, including partitioning of the system into well defined

components, their interconnection, and finally implementation assignment of each component. Thus, at this level timing is introduced into the system descriptions (at various granularity levels, e.g. bus-cycle accurate, clock accurate, etc.) and numeric formats are defined. This in turn allows more detailed simulations of each individual part of the system, leading to more precise evaluation of computation and communication loads. This new information allows the implementation of each component to be determined, either as a software component running on a standard processor core, or as a dedicated hardware component connected to a system bus. Other architectural choices can also be made as a result, such as the selection of suitable processor cores, number and type of system buses, as well as the size and structure of the memory hierarchy.

Finally, the implementation team tackles all the implementation details (of both hardware and software components) and their integration into a complete system. A number of system descriptions exists at this level, each individually suited to the implementation target, such as for example an ANSI-C description for a software component implemented on a DSP, or a Verilog netlist for a hardware component. System integration results in the hardware implementation of the System-on-Chip (SoC), as well as correct execution of all the software components of the system on this hardware platform.

2.1.4 Internal Barriers

Many potential barriers to design productivity arise from the design teams themselves, their organization, and interaction. These internal barriers include varying system specifications and models across different teams, as well as inherent differences in design skills in different teams.

As can be seen in Figure 2.2, feedback of information is an integral part of the development process. In particular, feedback of design errors (bugs) in the system is a critical

exercise. This information needs to be fed back as soon as possible, in order to minimize the cost of the error to the design process. It is estimated that if an error created at the algorithmic level and also corrected at this level produces a nominal cost of €1, it would then produce a cost of €6 if allowed to propagate to the architectural level, and a cost of €100 if allowed to propagate all the way to the implementation level [18].

Hence, it is clear that the efficiency of the design process, both in terms of time and cost, depends not only on the forward communication structures between teams, but also on the feedback structures in the design process. Several bottlenecks have been identified within both forms of communication between teams and are discussed here.

Firstly, from the onset, the design process faces a difficulty in communicating both design refinements (feedforward) and errors (feedback) due to the lack of a **unified** specification. Traditionally, different teams use separate and incompatible system specifications, thus ruling out any inter-team verification, both when the design is being handed off from one team to the next, and when any design errors are encountered.

Furthermore, the design teams use separate system descriptions. Additionally, these descriptions are very likely written in different design languages. In the forward direction, these factors make it impossible for the system description to be reused as it is passed from one team to the next. Hence, the system description has to be rewritten on each handover between teams. This results in **wasted design effort**, possible **misalignment between functionalities of the system descriptions** of the teams, as well as **additional coding errors** during recoding. In the feedback direction, localization and correction of design errors is not directly possible, as the information being fed back does not pertain to the system description where the error is to be corrected. This results in **unnecessarily prolonged debugging phases**.

Finally, each team possesses various unique skills, needed to tackle the wide range of

design issues that are encountered during the design process. While this variety of skills itself does not directly pose any difficulties and is indeed necessary for the completion of the design process, it can indirectly lead to inefficiencies. Primarily, the varying design skills across different teams lead to the fragmented tool support (discussed in detail in the following section), but can also further impede both forward and feedback communications between teams, due to varying foci of the respective system specifications and system descriptions within each team.

2.1.5 External Barriers

Numerous bottlenecks in the design process are caused directly by factors outside the design teams themselves. For the most part, these barriers come from the EDA software tools: both the individual tools themselves, as well as the general state-of-the-art in the EDA industry as a whole.

EDA Tools

Firstly, as already hinted to in Section 2.1.4, the work of separate design teams is supported by a wide array of different EDA software tools. Thus, each team uses a completely separate set of tools to any other team in the design process. Moreover, these tools are almost always incompatible, preventing any direct and/or automated cooperation between teams.

The root of this **fragmentation of the EDA tool support** for various teams lies, as already mentioned, in the variety of necessary skills found across different teams. Since EDA tools in general are written as commercial products and sold as support for design engineers, they are naturally clearly targeted to support the particular skills and responsibilities of each separate team.

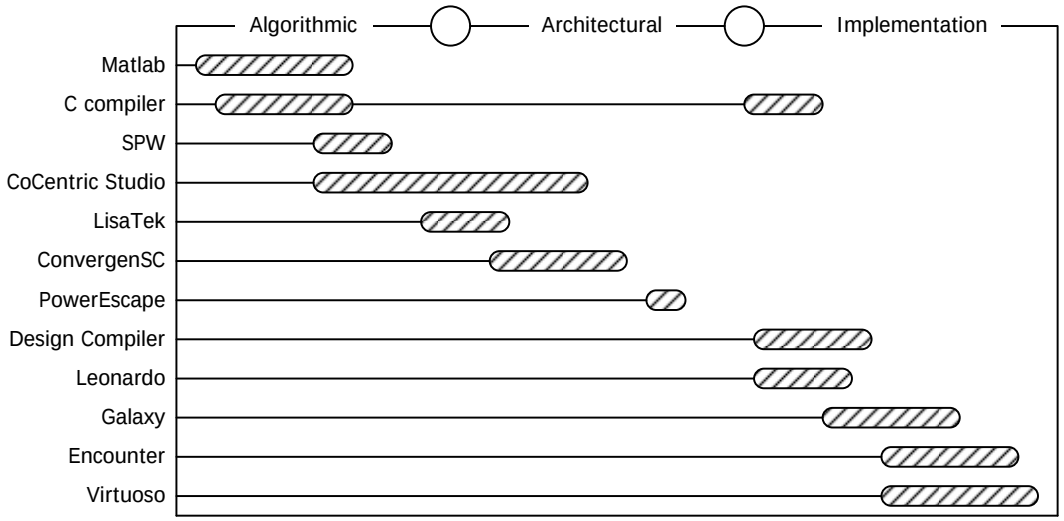

Figure 2.3: *Overview of the EDA tool support for the entire design process*

Hence, a survey of some of the most popular EDA tools currently available reveals the fragmentation of the automated support for the design process, as presented in Figure 2.3. It can immediately be noted that no single EDA tool supports the entire design process, from initial concept to final product. Moreover, no single EDA tool is able to support even one entire team completely (algorithmic, architectural, or implementation).

Furthermore, none of the presently available EDA tools on the market exhibit any explicitly built-in form of interoperability with tools of different vendors, which would enable seamless building of tool chains. A recent initiative by the SPIRIT consortium [4] aims to provide a mechanism for exchange of Intellectual Property (IP) blocks between EDA tools of different vendors. This would enable building of multi-vendor tool chains seamlessly. However, this initiative so far lacks a wide support base in the EDA industry and conceptually does not provide for a unified description of the system (for a more detailed discussion, see Section 2.3).

Also, EDA tool support exhibits several "gaps", i.e. parts of the design process which are critical, yet for which no automated tools are available. Although they have high

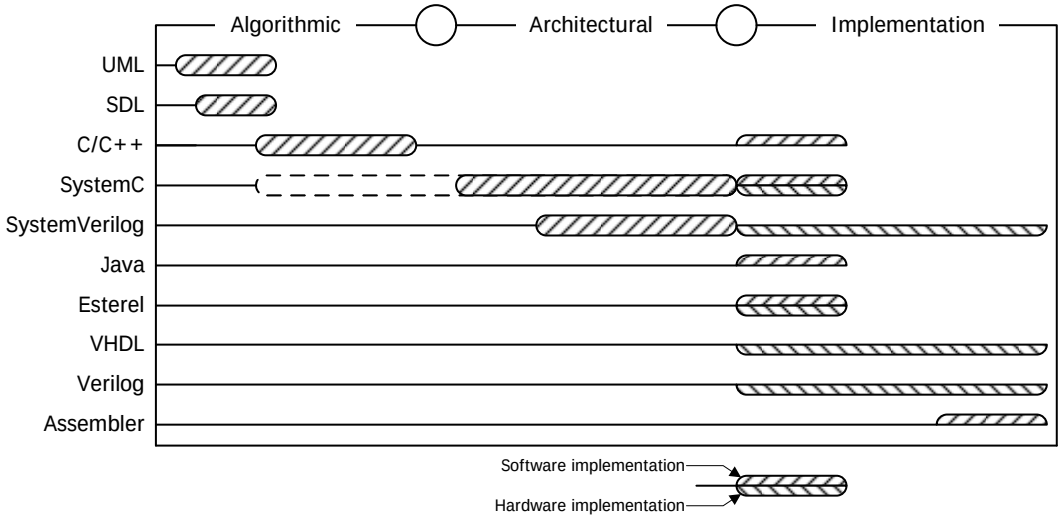

Figure 2.4: *Overview of the modeling language support for the entire design process*

impact on the rest of the design process, these steps typically have to be performed manually, due to their relatively high complexity, thus requiring designer intervention and effort. Designers typically leverage their previous experience to a large extent when dealing with these complex issues. Examples of such design steps are hardware/software partitioning, floating-point to fixed-point conversion, architecture mapping, and others. The necessity to perform these steps manually lowers the efficiency of the design process significantly.

Design Languages

Additionally, the fragmentation of the automated support of the design process is made more severe through the existence of numerous design languages. Similarly to EDA tools, design languages focus on a particular set of tasks within the design process and provide modeling constructs for it. Hence, each language supports only a portion of the design process, as shown in Figure 2.4.

From this diagram, it can immediately be noted that although relatively many design

languages support the algorithmic and implementation levels, there is little language support at the architectural level. Moreover, the language support at this level is only a recent development, as will be shown.

At the algorithmic level, languages like UML and SDL are available, providing modeling support even at very high levels of abstraction. Traditionally, systems are not formally modeled at these high levels, but the modeling starts when all the parts of the system and their interactions have to some extent already been explored analytically. Hence, the C and/or C++ languages have a more established base in algorithmic modeling, starting at a slightly lower level of abstraction.

It is interesting to note that the C/C++ languages are used in two parts of the design process. At the algorithmic level, they are used to model the behaviour of the system, i.e. the algorithm itself. At the implementation level, C/C++ is widely used for implementation of the software components of the embedded system, usually running on a standard processor.

A variety of other languages are available at the implementation level, each associated with a particular implementation option. Hardware components may be modeled in one of the hardware description languages (HDLs), of which VHDL and Verilog are the most popular. Hence, these languages exhibit constructs most closely suited to description of hardware systems. On the other hand, software components are typically modeled in C/C++, as already mentioned, or any of the other suitable languages like Java, or assembler. It should however be noted that the abstraction level offered in assembler is very low, and hence any large parts of code (such as entire applications) are not modeled well in this language. Nevertheless, the precise control which this low abstraction level brings, affords the designers the opportunity to write small, hardware-close pieces of software (such as device drivers) which are nearly optimal.

As mentioned previously, the availability of modeling languages at the architectural level is inadequate, especially compared to the support available at the other levels of the design process. What is more, the only design languages at this level, SystemVerilog and SystemC, are both relatively new developments and are still in the development phase.

SystemVerilog [19] is a natural extension to the Verilog HDL, extending the abilities of the core language into higher abstraction levels. In particular, many constructs of the C/C++ languages have been adopted. Hence, unlike Verilog, SystemVerilog possesses constructs for unions, structures, classes, inheritance, dynamic arrays, sophisticated loop control, and new data types, among others. With these enrichments, the aim of SystemVerilog is to help designers start describing the relatively abstract architectural structure through conceptual interfaces, rather than describing concrete, implementable functions connected through registers and implementable data types. This enables the designer to refine the abstract model down to the customary Register Transfer Level (RTL) design without having to migrate languages.

The strength of SystemVerilog is thus still based on Verilog's traditional competence at the RTL abstraction level, expanding this ability into architectural modeling. However, SystemVerilog fails to support the design process at the algorithmic level of abstraction, because of its inherent inability to efficiently describe and simulate typical algorithmic models which have very wide scope (often entire systems of systems), but low level of detail. Hence, using SystemVerilog to model such systems would result in extremely long simulation times.

On the other hand, SystemC [20] is based directly on the C/C++ languages and hence inherits their strengths both at the algorithmic and implementation levels, while attempting to create new support (which was so far missing) at the architectural level.

The SystemC language was divided from its conception into three parts: SystemC 1.x for modeling at the implementation level, SystemC 2.x for architectural modeling, and SystemC 3.x, for system level modeling. The latest available version of SystemC is version 2.1, released in June 2005. Hence, the coverage of SystemC in Figure 2.4 above the architectural level is dashed, to present ongoing development.

The first release of SystemC, version 1.0, was aimed at the implementation level of the design process, where the C/C++ core languages already hold a firm base. However, the crucial innovation brought about by SystemC 1.x at this level is the ability to model both software as well as hardware components, unlike any derivative of the C/C++ languages to date. Hence, it included new hardware-oriented modeling constructs such as concurrency, clocks, and well defined fixed-point datatypes.

SystemC 2.x raised the available level of abstraction by including constructs for modeling at the architectural level, also known as transaction level modeling (TLM). This second major release of SystemC abstracts **functionality** from **communication**, thus allowing seamless modeling of all parts of the design, regardless of whether their final implementation will be in hardware or software. This allows the designer to initially concentrate on architectural modeling, i.e. breaking up the system into appropriate modules, and defining the communication (which is not yet synchronous to a clock) between these modules. The designer can therefore focus on critical architectural issues, such as finding communication bottlenecks, estimating computational complexity of each individual module, and consequently defining the hardware/software partitioning of the system. When this architectural modeling is performed in SystemC 2.x, it is possible to refine the functionality of each module down to concrete software or hardware (RTL) implementation, as well as refine each communication interface down to an explicit implementation (e.g. a bus interface, a direct wire connection, or a shared memory block) without migrating to another language.

Future developments of SystemC, in version 3.x, will focus on the algorithmic and architectural levels of the design process. Currently under development are mechanism for modeling high-level real-time constraints of the system, concurrency of software parts of the system (multi-threading), and flexible hardware/software partitions, which are suitable for reconfigurable architectures.

The fragmentation of the EDA support described above, both in terms of tools and languages, leads to an inefficient structure of the design process. Due to the inherent incompatibility of EDA tools, especially from various vendors, manual effort is required in progressing a design through a whole tool chain. The presence of gaps in the EDA tool support necessitates filling them with manual effort as well. Finally, use of many different design languages throughout the design process necessitates numerous rewritings and reformattings of the system description, which is also largely a manual process. Naturally, all this manual effort lowers the efficiency of the design process significantly. Also, the quality of the design process is adversely affected through the introduction of manual coding errors. Finally, it should be noted that in many instances, all the forms of manual effort listed above can be replaced by either academic tools, or in-house developed automation (as opposed to commercial EDA tools) in various forms, such as scripts, templates, parameterized (i.e. reusable) designs, etc.

2.1.6 Summary

The rapid growth in algorithmic complexity of wireless communications standards, as well as the sustained growth in silicon complexity has created the algorithmic and design gaps. The existence and continuing widening of these gaps point to an urgent need for more efficient design methodologies.

Several bottlenecks in the design process leading to significant inefficiencies can be

identified. Within the design process itself, the use of separate system descriptions by each team is the primary bottleneck, causing time waste and degraded quality of design. Considerable loss of efficiency and quality is also caused by impeded communication of errors (bugs) during the design process.

External inefficiencies arise from factors outside the design process itself, and stem primarily from the fragmented EDA tool and language support. Incompatibility of EDA tools between various vendors, use of a variety of different design languages, and the presence of large gaps in EDA tools support are all causing severely reduced efficiency.

2.2 Solution Outline

In the previous section, a number of acute bottlenecks in the design process have been identified. It is the aim of this section to outline the requirements of the solution for relieving these bottlenecks.

In essence, an environment is needed which transcends the interoperability problems of modern EDA tools. In other words, this environment would allow building of tool chains in a fully automated, seamless fashion, although the tools themselves have not been designed to be interoperable. To achieve this, the environment will have to be flexible in several key aspects.

Firstly, the environment has to be **modular** in nature. This is required to **allow expansion** to include new tools as they become available, as well as to enable the designer to build a custom design flow only from those tools which he/she needs.

Also, the environment has to be **independent from any particular vendor's tools or formats**. Hence, the environment will be able to integrate tools from various vendors, as well as academic/research projects, and any in-house developed automation, such as

scripts, templates, or similar.

A further requirement is that the environment must be able to **operate over a wide range of design languages**. This should in principle include any of the algorithmic, architectural, and implementation languages described in Section 2.1.5. However, the modular nature of the environment should also allow for inclusion of any new languages as they become available.

To allow unobstructed communication between teams, the environment should eliminate the need for separate system descriptions. Hence, the single system description, used by **all** the teams simultaneously, would provide the ultimate means of co-operative refinement of a design, from the initial concept to the final implementation. Such a single system description should also be flexible through having a modular structure, accommodating equally all the teams. Thus, the structure of the single system description would be a superset of all the constructs required by all the teams, and the contents of the single system description would be the superset of all the separate system descriptions used by the teams currently.

The use of such a system description would eliminate the need for any rewriting of system descriptions (as described in Section 2.1.4), thus leading to drastic improvements in efficiency and quality of the design process. Additionally, the feedback of errors caught during the design process would be significantly more efficient, through the immediate focus of the feedback information, due to the use of only one system description throughout the design process.

2.3 Related Work

The inefficiency imposed by the lack of interoperability between modern EDA tools is a known problem which is currently attracting considerable research interest and the first

academic and commercial solutions to this problem are starting to surface.

The SPIRIT consortium [4] acknowledges the inherent inefficiency of interfacing incompatible EDA tools from various vendors, as already mentioned in Section 2.1.5. The work of this international body focuses on creating interoperability between different EDA tool vendors from the point of view of their customers, the product developers.

Hence, the solution offered by the SPIRIT consortium [21] is a standard for packaging and interfacing of IP blocks used during system development. The existence and adoption of this standard ensures interoperability between EDA tools of various vendors, as well as the possibility for integration of own IP blocks which conform to the standard.

However, this approach requires widest possible support from the EDA industry, which is currently lacking. Also, even the full adoption of this IP interchange format does not eliminate the need for multiple system descriptions over the entire design process. Finally, the most serious shortcoming of this methodology is that it provides support only for the lower levels of the design process, namely the lower part of the architecture level (component assembly) and the implementation level.

Another notable approach to EDA tool integration is provided by the Model Integrated Computing (MIC) community [5]. This academic concept of model development gave rise to an environment for tool integration [22].

In this environment, the need for centering the design process on a single description of the system is also identified, and the authors present an implementation in the form of an Integrated Model Server (IMS), based on a database system. The structure of the entire environment is expandable and modular in structure, with each new tool introduced into the environment requiring a new interface.

The major shortcoming of this environment is its dedication to development of software

components only. As such, this approach addresses solely the algorithmic modeling of the system, resulting in software at the application level. Thus, this environment does not support architectural and implementation levels of the design process, which are also within the scope of this thesis.

Synopsys is one of the major EDA tool vendors offering automated support for many parts of the design process. Recognizing the increasing need for efficiency in the design process and integration of various EDA tools, Synopsys developed a commercial environment for tool integration, the Galaxy Design Platform [23].

This environment is also based on a single description of the system, implemented as a database and referred to as the open Milkyway database. Thus, this environment eliminates the need for rewriting system descriptions at various stages of the design process. It also covers both the design and the verification processes and is capable of integrating a wide range of Synopsys commercial EDA tools. An added bonus of this approach is the open nature of the interface format to the Milkyway database, allowing third-party EDA tools to be integrated into the tool chain, if these adhere to the interface standard.

However, this environment is essentially a proprietary scheme for integrating existing Synopsys products, and as such lacks any support from other parties. In fact, as at the end of 2005, only Synopsys tools are integrated into the Galaxy Design Platform. An ever more serious limitation of this approach is the lack of support for any other part of the design process other than the implementation level, and then only for hardware components.

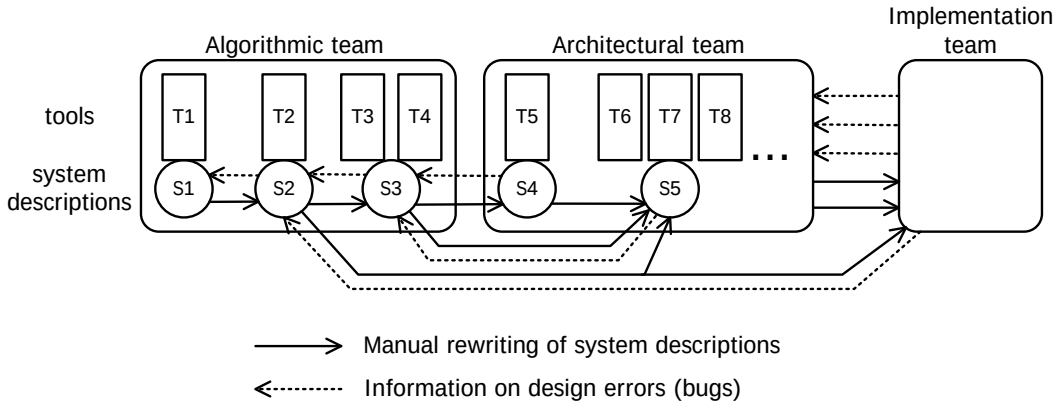

Figure 2.5: *Example of constructing a traditional tool chain over the three teams and several tools and system descriptions*

2.4 Proposed Framework

Traditionally, tool chains in modern design processes are constructed from a number of EDA tools used by each of the three design teams. These tools operate on a number of system descriptions, maintained by the appropriate team members. In some instances, more than one tool operates on one system description. This is illustrated in Figure 2.5, where system descriptions are represented by circles, and EDA tools that operate on them are represented by rectangles.

As the process of design refinement goes on, the design is handed over from one team to the next by passing the required system descriptions from the first team, which are then manually rewritten to form system descriptions used by the second team. In the example tool chain in Figure 2.5, system descriptions S2 and S3 are passed from the algorithmic team to the architectural team (indicated by continuous lines). It can be noted that the architectural team uses only the system description S3 for creating the system description S4, while creating the system description S5 necessitates both the system descriptions S2 and S3.

Similarly, information on design errors (debugging information) needs to be passed back-

wards through the tool chain. In the example in Figure 2.5, all the information found on an error in the system description S4 is passed back to the system description S3 for correction. Also, all debugging information from the system description S5 needs to be fed back to the system description S3 as well.

Hence, interdependencies exists among the various system descriptions of each individual design team, but even more importantly among the different teams as well. Thus, in the given example, the implementation team requires two system descriptions from the architectural team and one from the algorithmic team in order to create its own system descriptions. Also, the implementation team feeds back three types of debugging information to the architectural team and one to the algorithmic.

In other words, the example tool chain in Figure 2.5 can be seen as one specific instance of the general structure of the design process shown in Figure 2.2, with all individual interdependencies between teams (both in the forward and in the feedback directions) being shown.

In contrast to this traditional structure of the design tool chain, this thesis proposes an **Open Tool Integration Environment** (OTIE), which complies with all the requirements set out in Section 2.2. Thus, the OTIE is based on the **Single System Description** (SSD), used by all the three teams in the design process. In other words, all the tools used by all the teams operate on just one description of the system, as depicted in Figure 2.6.

In the OTIE, each tool in the design process still performs its customary function, as in the traditional tool chain, but the design refinements are now not stored in separate system descriptions and thus subject to constant rewriting. Rather, the OTIE provides its core, the SSD, as a consistent repository for all refinement information. Thus, the SSD is a superset of all the system descriptions present in the traditional tool chain.

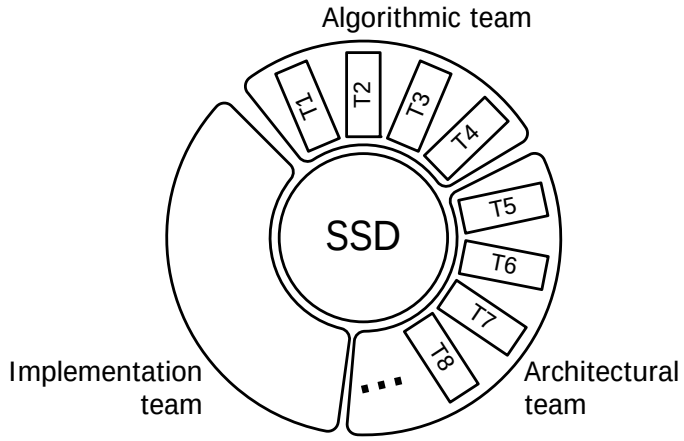

Figure 2.6: *The proposed tool chain, based on just one system description (SSD) and incorporating the same tools as the traditional tool chain*

Hence, throughout the design process, design refinements are accumulated in the SSD. At the beginning of the process, the empty SSD is filled with the initial algorithmic model of the system, which is then refined through the customary succession of design teams and EDA tools they employ. Each of these steps in the design process enriches the description of the system, adding important design details to the SSD. At the end of the design process, the SSD contains all the refinement information gathered by all the teams during the design process. For example, this may include all the textual and graphical representations of the system, the code in all the design languages used during the process, the revision history of all the parts of the design, the test cases used during verification, the simulation traces, and so forth.

An important aspect of the OTIE is that from the viewpoint of all the designers, in all the three teams, the use of EDA tools remains completely unchanged, compared to the traditional tool flow. In other words, adopting a centralized repository such as the SSD does not force the designers to either change the EDA tools they use, or the way they use them.

Each of the tools used in the design process has its own **view** of the SSD, i.e. the

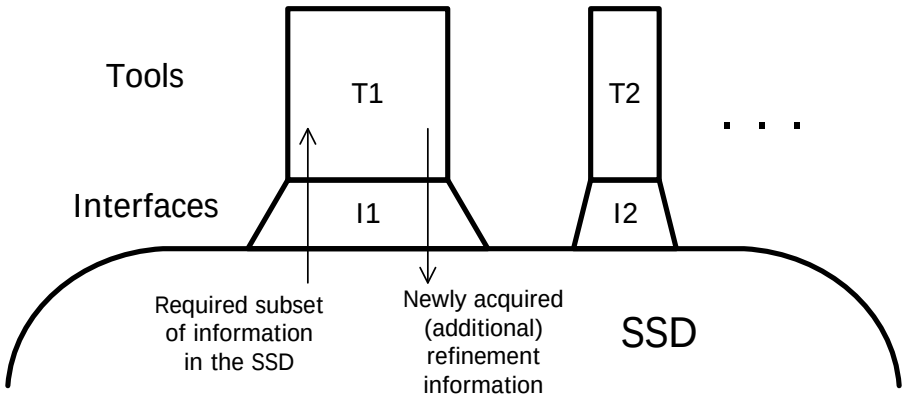

Figure 2.7: *EDA tools T1 and T2 are integrated into the OTIE through the interfaces I1 and I2 respectively*

subset of the SSD which is used by the tool, as well as the information which is to be updated or added to the SSD by the tool. The presentation of such a view to the EDA tool, including the storage of all the new design refinements introduced by this tool, is performed by the tool's **interface**.

Therefore, the modularity required in the OTIE is achieved through provision of appropriate interfaces for each individual EDA tool which is to be integrated into the environment. This is shown in Figure 2.7, where the tools T1 and T2 have been integrated into the OTIE through the interfaces I1 and I2 respectively, each providing the corresponding tool with its required view of the SSD.

Under the OTIE, all the individual design team members continue to use the appropriate EDA tools in exactly the same manner as in the traditional tool chain. In other words, the existence of the SSD and the structuring of the EDA tools into a consistent design environment is transparent to the designers.

Where gaps in the design flow exist, as discussed in Section 2.1.5, the modular structure of the OTIE allows integration of both research tools and in-house developed automation, in the same fashion as commercial EDA tools.

Additionally, the OTIE provides tools for importing textual system descriptions created by the designers, directly into the SSD. These System Description Interfaces (SDIs) form an integral part of the OTIE. Similarly, the OTIE is able to directly export the status of the SSD to the designer, in various formats, through so-called Visualizers.

2.5 Implementation

In addition to fulfilling the requirements set out in Section 2.2, the environment proposed in this thesis also breaks up the tasks of implementing a suitable framework into several well-defined pieces. From the discussion in Section 2.4, these implementation tasks can be identified as follows.

- **SSD**

  The core of the OTIE.

- **SDIs**

  The tools for direct importing of textual system descriptions.

- **Visualizers**

  The tools for extracting and visualizing the information in the SSD.

- **EDA tool chains**

  The various existing EDA tools (or chains of them), each of which automates a particular part of the design process.

Thus, the implementation of each of these elements is discussed in turn in the following sections.

2.5.1 Single System Description (SSD)

As outlined in Section 2.4, the core of the OTIE is the SSD, a single repository for all the design refinement information throughout the entire design process. From these propositions arise several implementation requirements for the SSD.

- **Extendibility**

  The amount of refinement information stored in the SSD increases rapidly as the design process goes on and the SSD implementation must allow for this growth.

- **Cooperation**

  All team members must have simultaneous read and write access to the contents of the SSD.

- **Security**

  Since all the design information is held in one place, the security offered by the SSD implementation is of key importance, especially in the areas of access privileges and storage of the revision history.

- **Access**

  The SSD implementation must allow for simple and fast access to the refinement information within, so that efficient interfaces can be built rapidly.

This thesis presents an implementation of the SSD based on a MySQL [24] database, fulfilling all the above requirements.

Firstly, the database implementation of the SSD supports virtually unlimited expandability, both in terms of structure and volume. As new refinement information arrives to be stored in the SSD, it can either be stored within the existing structure (sheer increase in data volume), or it may require an extension to the entity-relationship structure of

the SSD. In the latter case, the structure of the database implementation of the SSD can easily be expanded through addition of new tables or links between tables. Hence, the expandability of this implementation is only limited by the hardware resources of the computer system hosting the database server.

Also, the database this implementation of the SSD is based on, is inherently a multi-user system, allowing transparent and uninterrupted access to the contents of the SSD to all the designers simultaneously.

Furthermore, the security of the database implementation of the SSD is two-fold. Firstly, the database system allows for detailed setting of access privileges of each team member and integrated EDA design tool to each part of the SSD. Hence, only the appropriate (i.e. authorized) tools and/or designers have access to any piece of refinement information in the SSD. Secondly, the database implementation also allows for seamless integration of a version control system (such as CVS [25] or ClearCase [26]), to automatically maintain revision history of all the information in the SSD.

Finally, accessing the refinement information (both manually and through automated tools) is greatly simplified in the database implementation of the SSD by its Structured Query Language (SQL) interface. This allows for efficient writing of tool interfaces (see Figure 2.7) and other tools which operate directly on the SSD.

It is also worth noting that several separate design processes (i.e. projects) can be supported simultaneously by a single MySQL server. Since each server can support multiple databases simultaneously, and each project requires a single database to implement its SSD, several projects can transparently share the single MySQL server, thus increasing the efficiency of using this resource.

2.5.2 System Description Interfaces (SDIs)

As was shown above, SDIs are needed to perform direct importing of textual system description written by designers, into the SSD. Since a number of design languages are currently in wide-spread use (see Section 2.1.5), a separate SDI needs to be developed for each design language. Also discussed in Section 2.1.5 is the advent of SystemC as a promising new addition to system design languages, covering a relatively large portion of the design process.

Hence, in this chapter an SDI for SystemC is presented as a proof of concept. Additionally, in the following chapters two complete SDIs will also be presented in use within complete tool chains in the OTIE: in Chapter 3 (Cossap SDI) and Chapter 4 (SystemC SDI).

Please note that the SDI concepts presented in this chapter are not **unique** only to the SystemC SDI. Rather, these concepts are as much as possible replicated in **all** the SDIs within OTIE. However, at the same time, the flexibility of OTIE is utilized to allow each separate SDI the possibility to extend the concepts presented in this section and store in the SSD additional refinement information which may be relevant only to the particular language in question.

Since many SDIs have to be written, their implementation in general should be aimed at high modularity and ease of reuse of components. Hence, the SystemC SDI presented here is built from two parts, as shown in Figure 2.8.

The front end of the SDI, the **parser** module, operates directly on the textual system description created by the designer, and is thus language-dependent. However, the back end, or the **scanner** module, operates only on the intermediate description of the system, and is thus language independent. Therefore, the front end of an SDI has to be written

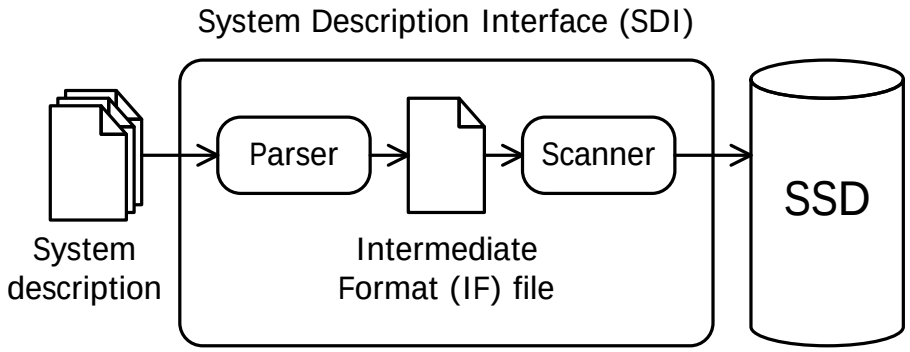

Figure 2.8: *The modular structure of the SystemC SDI*

completely new for each design language, whereas the back end can be reused without modifications.

The communication between the two modules of the SDI is achieved through an Intermediate Format (IF) representation of the system, which captures the design information extracted by the parser module, before it is stored in the SSD by the scanner module. The IF representation is formatted in the eXtensible Markup Language (XML) due to its good human readability and high popularity, which in turn resulted in many XML software tools (such as syntax checkers, parsers, editors, etc.) being available. Also, basing the IF representation on XML allows for unobstructed expansion of the semantic set (e.g. for adaptation to a new design language in the future) simply by defining additional XML tags.

The functionality of the scanner module is to enrich the SSD with the design information in the XML IF representation of the system. Therefore, the semantics in the XML IF need to match the entity-relationship structure of the subset of the SSD which will hold this information. In other words, the newly acquired design information, held in the XML IF, needs to be able to fit into the SSD. The entity-relationship structure of the subset of the SSD that holds the information supplied by the SystemC SDI is shown in Figure 2.9.

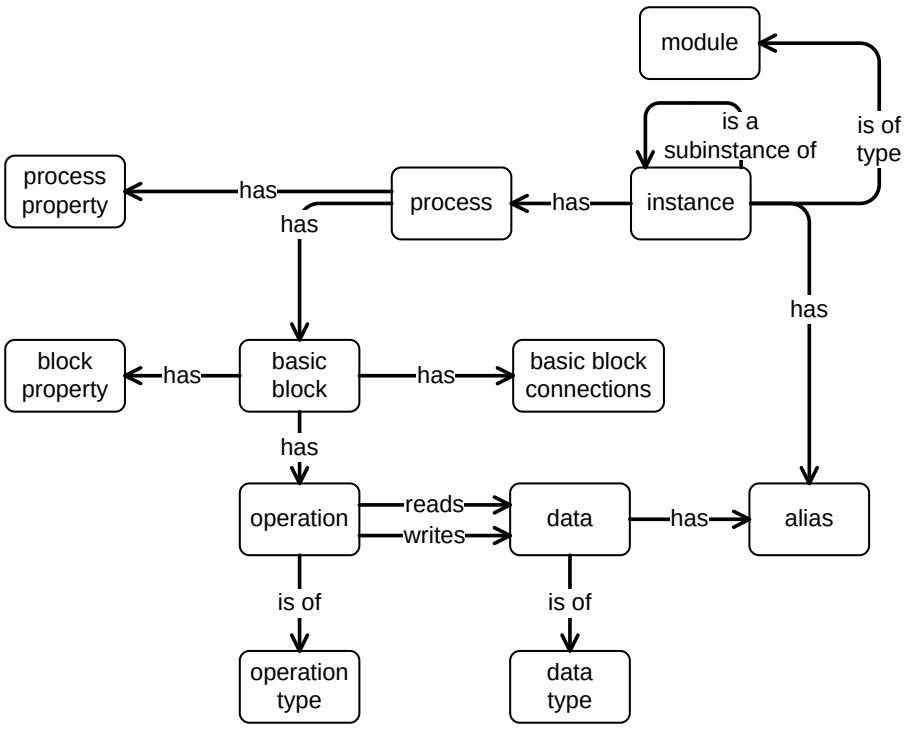

Figure 2.9: *The entity-relationship structure of the subset of the SSD holding the information from the SystemC SDI*

Here, the boxes represent **entities**, or types of information contained in the database. The arrows represent **relationships** among the different entities in the database. An entity can thus be considered a "noun", such as **data**, or **data_type**, and a relationship can be considered a "verb", such as *has*, or *is of type*. Entities are represented in the form of database tables, with columns and rows, much like a spreadsheet, while relationships are represented in the form of links, connecting the information in one table to the information in another.

The entity **module** in Figure 2.9 represents the break-down of the system functionality by the designer, using the SystemC SC_MODULE classes. Through modules, the designer represents the system with each individual part of its functionality contained in a separate module, with its own inputs and outputs.

Actual use of modules in the SystemC model of the system is made by instantiation, i.e. the declaration of at least one particular **instance** of that module, with connections to other instances in the design through existing data channels. Hence, instance is another entity in the structure of the subset of the SSD covering the SystemC SDI. Instantiation is represented in Figure 2.9 by the *is of type* relationship of every instance entity to a module entity. It is worthwhile noting that more than one unique instances of each module may exist in the overall representation of the system, but a module declaration which is not instantiated at least once, is redundant.

There are generally two types of instances in SystemC: structural and functional. Structural instances contain no explicit functionality, but do contain instances of other modules within them, thus allowing for the existence of a hierarchy in the SystemC description. On the other hand, functional instances do not contain any sub-modules (and are hence always the leaf nodes of the hierarchy tree), but they do contain explicit functionality.

This hierarchy of the SystemC description is represented in Figure 2.9 through the *is a sub-instance of* relationship of one entry in the instance table to another. Hence, all instances in this table are linked up into a single hierarchy tree, whose root is referred to as the top-level instance, containing the entire design.

Embedded in the functional instances, as already stated, is the functionality of the system. Each functional instance contains one or more **processes**, as represented in Figure 2.9, connected to the appropriate instance through the *has* relationship. While all the processes throughout the design are running concurrently, each process is in itself purely sequential, i.e. contains a Control-Flow Graph (CFG) representation. The CFG representation of each process shows the sequential progress through the execution of the process. The CFG is built up of nodes, each of which is a so-called **basic block**,

which is also represented as an entity in Figure 2.9, linked from its parent process by a *has* relationship.

The structure of the CFG of each process is represented through a detailed list of predecessor and successor nodes for each basic block. Through this technique, it is possible to link up basic blocks into any arbitrary CFG structure as needed. The lists of each basic block's predecessors and successors is contained in the **basic block connections** entity.

Each basic block is itself a graph - Data Flow Graph (DFG) - whose each node is an **operation**. Thus, each basic block by itself contains no control flow, that is to say, the execution of all the operations within each basic block is always the same. Operations are atomic, i.e. are not further divisible, and are of a certain **operation type**.

Additionally, process and basic block entities may contain several properties. The entity **process property** may for example contain information on the longest path through the CFG of the process, or γ, the degree of parallelism in the CFG. The entity **block property** may for example contain information on the depth of the DFG of the basic block, or the number of addition operations in the basic block. Such property information is essential in performing high-level system characterization through static code analysis [27].

Communication between instances in a SystemC description is achieved through data channels, represented in Figure 2.9 by the **data** entity. Each data entity is bound to one **data type** entity, through a *has* relationship.

Since a single data channel, on the way from its source to its destination, can traverse several instances in the hierarchy of the design (as depicted in Figure 2.10), it has a number of different aliases referring to it. Hence, **alias** is another entity in the structure of the subset of the SSD holding the SystemC SDI information, as shown in Figure 2.9.

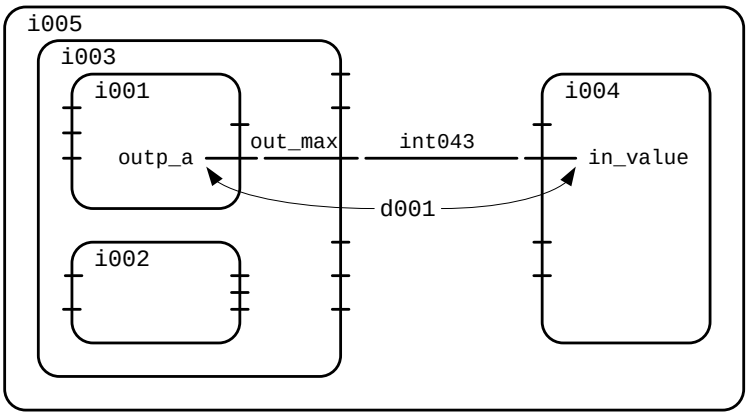

Figure 2.10: *Traversal of several instances by a single data channel, showing its various aliases*

Figure 2.10 shows a data channel `d001` traversing four instances, `i001`, `i003`, `i004`, and `i005`. Under the scope of each individual instance, this data channel has a separate alias. Hence, at its source, the instance `i001`, it has the alias `outp_a`. In the instance `i003` it has the alias `out_max`, in `i005` it is referred to as `int043`, and at its destination, `i004`, it has the alias `in_value`.

The *has* relationship between the data and alias entities represents the different aliases one data channel may have throughout the hierarchy of the design. Similarly, an instance entity is bound to a number of different alias entities through the *has* relationship, representing all the different aliases for all the data channels within that particular instance.

Finally, two relationships exist between data channel and operation entities. The *reads* relationship represents the use of a data channel as an input into one or more operations. Similarly, an operation has the relationship *writes* to one or more data channels. It is important to note that each data channel may be read by one or more operations, and each operation may read one or more data channels. On the other hand, each data channel is written by exactly one operation (and no more than one), though an operation

may write one or more data channels.

As already stated, the semantics of the XML IF representation of the system agree closely with this entity-relationship structure within the SSD. The listing of this XML format is provided in Appendix C, together with a short example.

The presented SystemC SDI demonstrates the concept of including a tool for directly importing textual system descriptions into the SSD. As such, it can be used within any tool chain integrated into the OTIE. As already mentioned, the use of two SDIs are demonstrated in Chapter 3 (Cossap SDI) and Chapter 4 (SystemC SDI).

2.5.3 Visualizers

As already mentioned at the beginning of Section 2.5, similarly to the SDIs, there exists a need for direct exporting of the contents of the SSD to the designer, through visualizers. These tools help the designers in monitoring the status of the SSD, as well as help visually summarize the relevant refinement information in the design.

One of the most suitable formats for visualizing the contents of the SSD is the Hyper Text Markup Language (HTML). This language is flexible, through its ability to describe any number of mutually linked pages. It is also suitable for displaying both textual and graphical information, which is a critical requirement for the visualization of the SSD. Finally, HTML is highly suitable because its viewing is ubiquitously supported by any web browser.

The HTML visualizer presented in this thesis is capable of representing all of the information in the SSD at any stage in the design process. In other words, it is general and dynamic in nature, adapting to the current contents of the SSD as it grows during the design process. Hence, it displays the contents of all the tables in the SSD textually,

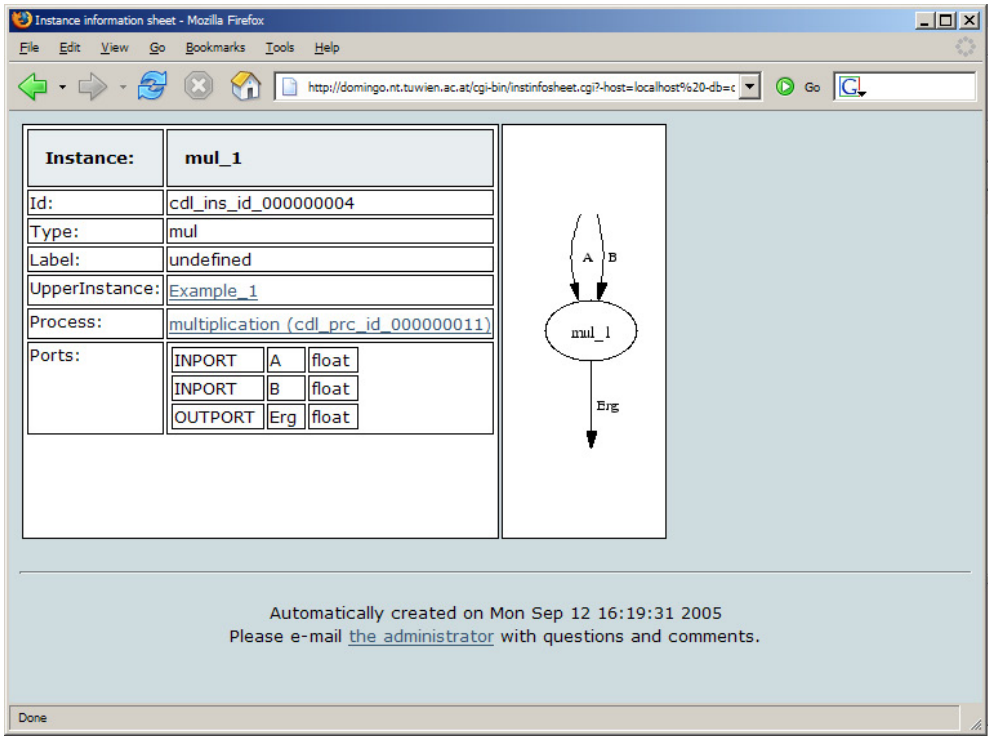

Figure 2.11: *Screen shot of the HTML Visualizer*

and whenever possible, augments this with graphical representations of the data.

A screen shot of the HTML visualizer is given in Figure 2.11. This screen shot shows the information page of a particular instance, showing its name (`mul_1`), place in the hierarchy, aliases it contains, its unique ID within the SSD, and so on.

The HTML visualizer is also capable of graphically representing the hierarchy tree of the design. To achieve its dynamic flexibility mentioned earlier, the HTML visualizer relies on creating dynamic HTML pages using Common Gateway Interface (CGI) scripts written in the Perl scripting language. These scripts are also capable of directly querying the MySQL database in which the SSD is implemented, thus providing a direct link from the SSD to the created visualization.

In this way, the HTML visualizer creates one of the possible direct exports of the refinement data contained in the SSD into a lucid visual form which gives the designer a clear

overview of the entire system.

2.5.4 EDA Tool Chains

Entire EDA tool chains, automating a specific part of the design process, can be integrated into the OTIE. These tool chains can include commercial EDA tools, academic projects, as well as in-house developed automation.

The integration of two separate EDA tool chains is presented in this thesis. The first, presented in Chapter 3, is an environment for automated development of Virtual Prototypes (VPs). It includes commercial tools, as well as research-developed software. The second tool chain is aimed at closing one of the gaps in the automated tool support for the design process, as discussed in Section 2.1.5. This environment, presented in Chapter 4, is called *fixify* and provides automated support for floating-point to fixed-point conversion, necessary as the algorithmic model of the system is refined towards implementation.

2.6 Conclusions

This chapter presented OTIE, a unique, extensible, and open environment for integration of EDA tools. OTIE is capable of integrating EDA tools from various vendors, as well as non-commercial EDA tools (academic or in-house developed). The core of the OTIE, the SSD, is based on a database system, making it a safe, performant, and flexible implementation. The OTIE is language-independent and hence compatible with any of the many design languages in use today. It is also capable of supporting the entire design process, from the initial concept to the final product, both in the design and verification flows, and for both hardware and software components of the system.

3. VIRTUAL PROTOTYPING

One of the most promising new design techniques, with great potential for increasing the efficiency and consistency of embedded system design flows, is virtual prototyping. An environment for virtual prototyping has successfully been created and integrated into the OTIE, in both the model development and verification flows. This chapter covers the concept of virtual prototyping, the structure and integration of the virtual prototyping environment in OTIE and its extension into the verification flow.

3.1 Concept of Virtual Prototyping

System descriptions at algorithmic level contain no specific implementation details. Hence, before implementation of the system can begin, the algorithmic description is partitioned, i.e. each component in the description is assigned to eventual software or hardware implementation.

Traditionally, implementation of hardware components proceeds from this point. Development of software modules, however, can begin only once all required hardware design is complete. This is due to the fact that the design of software components must take into consideration the behaviour of the underlying hardware. Hence, a significant penalty is incurred in the length of the design process (see Figure 3.1, top chart).

Virtual prototyping is a technique which can eliminate most of this penalty and thus dramatically shorten the development cycle [28, 7]. A Virtual Prototype (VP) is a

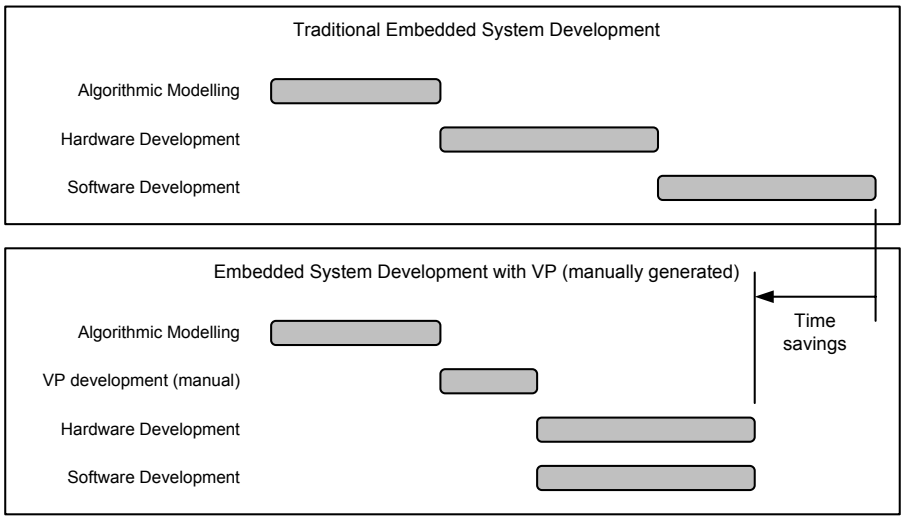

Figure 3.1: *Shortening of the design cycle by the VP technique*

software model of the complete system, fully representing its functionality, without any implementation details. To achieve the mentioned system development speedup, we consider VPs which additionally include full definitions of hardware/software interfaces found in the system, including the required architectural information, but still no details of the actual implementation of any component.

The speedup in the system development cycle by employing virtual prototyping is achieved as depicted in Figure 3.2. Firstly, the algorithmic model is partitioned into components to be implemented in hardware and those to be implemented in software. This defines the hardware/software interfaces in the system. In Figure 3.2, blocks B, C, and E have been assigned to implementation in hardware and blocks A, D, and F in software. The algorithmic description is then remodeled to a form where these interfaces are clearly defined. This definition of inter-component interfaces includes high-level definitions of buses in the system architecture, direct I/O ports, as well as the specification of shared register or memory spaces through which communication between components takes place. Thus, the VP of the system is created.

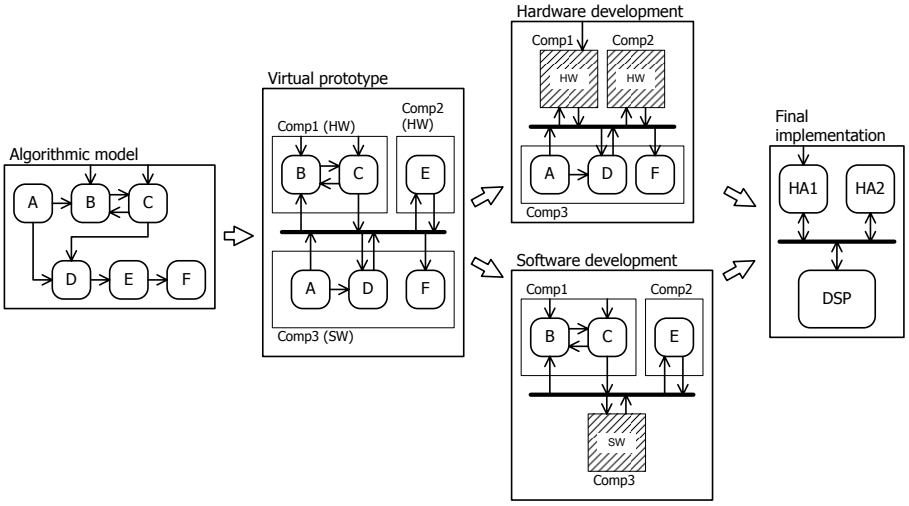

Figure 3.2: *System development using a VP*

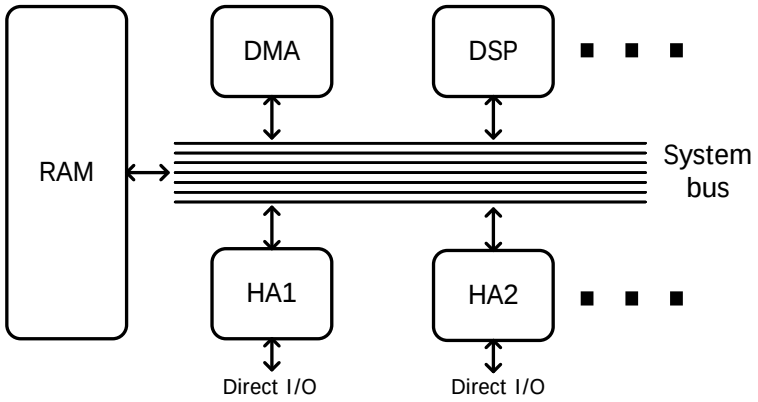

Figure 3.3: *Model of the hardware platform*

From this point, hardware and software development proceed in parallel. It is important to note that all blocks assigned to hardware implementation are grouped into a number of VP components, each of which will later be realised as a separate Hardware Accelerator (HA) in the system architecture (see Figure 3.3). In Figure 3.2, blocks B and C form the VP component `Comp1`, whereas block E alone forms the VP component `Comp2`. The blocks A, D, and F are implemented as software components, running on the DSP in Figure 3.3.

Development of the hardware implementation of VP component `Comp1` is achieved against the hardware/software interface defined in the VP. Similarly, the hardware implementation of VP component `Comp2` relies on the existence of the same hardware/software interface. At the same time, the development of the software implementation of VP component `Comp3` makes use of the same interface. Such use of the VP ensures co-operability of the three implementations, allowing for their parallel development and the resulting time-savings.

Virtual prototyping offers numerous improvements to the design process. First and foremost, it allows parallel development of **all** components in the system, resolving all interface dependencies. Furthermore, it allows verification of software components which interface with hardware against the known hardware/software interface. Finally, a VP allows verification of the hardware implementation itself, making sure the hardware indeed provides correct interface to external components as it was designed for at the algorithmic level.

Very importantly, creation of a VP for a system component requires a relatively small design effort, compared to that of a full hardware or software implementation. This is due to the relaxed requirement of the VP to recreate behaviour only at component boundaries, allowing all other implementation details to be overlooked. As seen in

Figure 3.1 (bottom chart), this allows the time savings which make VP an efficient design technique.

3.2 Related Work

Previous work on VPs has for the most part focused on their use, in the hardware/software co-simulation of the embedded system [29, 30, 31]. While these efforts are targeted towards increasing the efficiency and quality of the design process through novel modifications of the co-simulation process, they ignore the significant gains achievable by automatic generation of VPs.

The approach presented in [32] considers automatic generation of VPs and achieves a speedup in the order of 5 to 8 times that of manual VP creation. However, this VP generation environment requires, in addition to the algorithmic description of the component, a formal description of its GLObal Control, Configuration and Timing (GLOCCT). Furthermore, this design environment considers strictly Synchronous Data Flow (SDF) models. The VP components automatically generated by this environment have GLOCCT code implemented in VHDL and the DSP code implemented in C.

The virtual prototyping environment integrated into the OTIE is based on a simulation framework defined by the VSI Alliance (VSIA simulation interface standard) [33]. This framework is highly suitable for simulation of VPs because of its simple (non event-driven) scheduling, which is adequate for this application and produces superior simulation performance.

In addition to SDF models, the virtual prototyping environment presented here supports Dynamic Data Flow (DDF) models. This enables automatic generation of VPs with dynamically configurable sample rates, both on the inputs and the outputs of each sub-module of a VP component. Functional invocations of each of the sub-modules of a

VP component in DDF models is also adaptable, changing with the availability of data flowing through the model. Also, the environment presented here eliminates the need for a separate GLOCCT description and implements the entire VP homogeneously, in automatically generated VSIA compliant C++ code.

Extension of the virtual prototyping environment into the verification flow requires automated verification pattern refinement, as explained in Section 3.4. Several previous research efforts in this area exist. Varma et al. [34] present an approach to reusing pre-existing test programs for virtual components. This approach includes a fully automated re-use methodology, which relies on a formal description of architectural constraints and produces system-level test vectors. However, this approach is applicable only to hardware virtual components.

On the other hand, Stöhr et al. [35] present FlexBench, a fully automated methodology for re-use of component-level stimuli in system verification. While this environment presents a novel structure which supports verification pattern re-use at various abstraction levels without the need for reformatting of the test patterns themselves, this in turn creates the need for new "driver" and "monitor" blocks in the environment for every new component being verified. Also, this environment has only been applied to hardware components.

An automated testing framework offered by Odin Technologies called Axe [36] also offers automated re-use of verification patterns during system integration. However, this environment requires manual rewriting of test cases in Microsoft Excel and relies on the use of a third-party test automation tool on the back end. Also, the Axe framework has only been applied to development of software systems.

The verification extension of the virtual prototyping environment presented here is also designed to provide fully automated verification pattern refinement, but addresses this

issue in a more general manner than previously published work. Hence, it is applicable to both software and hardware components, and indeed to verification pattern refinement between any two abstraction levels, though the particular instance of this framework presented here is specific to the transition from algorithmic to virtual prototype abstraction levels.

3.3 Generation of Virtual Prototypes

As described earlier, design of an embedded system proceeds from the algorithmic-level description towards the system's final implementation firstly through a partitioning process, followed by the creation of a VP and finally hardware or software implementation of each individual component.

The process of VP generation is typically performed manually, through rewriting of the VP from the algorithmic-level description. However, when the VP design environment is integrated into a unified design methodology such as OTIE, it is possible to make VP generation a fully automated process. This helps eliminate human errors and drastically decrease the time needed to create a VP, in turn deriving maximum possible efficiency gain promised by virtual prototyping [7, 37]. This is illustrated in Figure 3.4.

The automatic VP generation environment presented here is depicted in Figure 3.5 and has been shown to save hundreds to thousands of person hours compared to manual VP creation [7]. The process of automatically generating a VP component from that component's algorithmic description consists of two parts. First, the algorithmic description of the entire system (encompassing all its components) is read into the Single System Description (SSD). This also includes partitioning of the system by labeling each system component for implementation in hardware or software. The second step in the process is the generation of all parts of the VP component from the SSD.

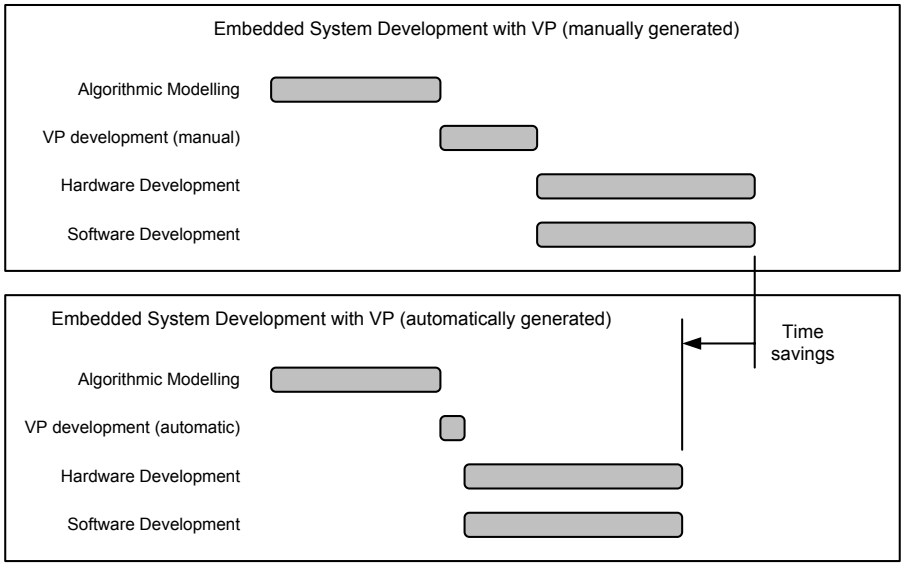

Figure 3.4: *Shortening of the design cycle by automating VP generation*

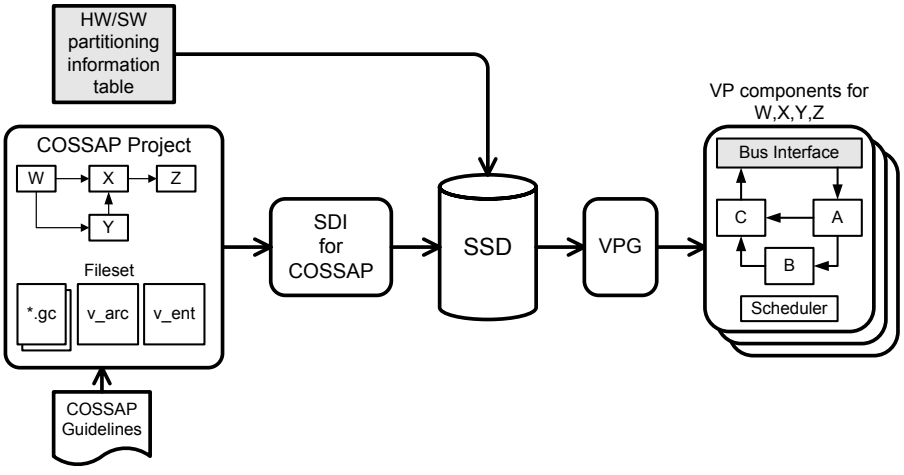

Figure 3.5: *Design environment for automatic generation of VPs*

3.3.1 Processing the Algorithmic Description

The environment for automatic generation of VPs presented here is based on processing algorithmic descriptions created in the Cossap environment. Nevertheless, the VP environment is in principle independent of languages and tools used for algorithmic modeling and can, due to its modular structure, easily be adapted to any language or tool.

Cossap descriptions contain separate structural (interconnection) and functional information. The structural and interconnection information in the Cossap description is VHDL-compliant, contained in the `*.v_ent` and `*.v_arc` files, and is read into the SSD by the System Description Interface (SDI), as shown in Figure 3.5. The SDI comprises a VHDL-compliant Parser module, as well as a Scanner module which manages the database structure within the SSD.

The functional information in Cossap descriptions is written in GenericC, which is an extension to ANSI C proprietary to the Cossap environment, contained in the `*.gc` files, and has to be formatted in accordance with specific guidelines (see Figure 3.5). These guidelines ensure compatibility of the GenericC code with tools in the second phase of the automatic VP generation process. Suitably formatted functional component descriptions are placed directly into the SSD. Please see Appendix D for a complete example of a Cossap design, including `*.v_ent`, `*.v_arc` and GenericC files.

After the complete algorithmic system description is processed into the SSD, it is necessary to perform hardware/software partitioning before VP components for all hardware components can be generated. Manually created hardware/software partitioning decisions, stored in textual form, are integrated directly into the SSD (see Figure 3.5). Also, possibilities for automated hardware/software partitioning exist and have been successfully applied to the presented environment [38], yielding the same quality of results as manual system partitioning. Once system partitioning has been performed, the first

phase of the VP generation process is complete.

3.3.2 Assembling the Virtual Prototype

A VP component is composed of several parts, as shown in Figure 3.5. The core of the automatically generated VP component is the recreated interconnected block structure – blocks A, B, and C in Figure 3.5 – as found in the algorithmic level model inside each component (components W, X, Y, and Z in Figure 3.5). Additionally, the VP component contains a scheduler which controls the execution of each block. The scheduler monitors the data passing between the blocks and executes a block when sufficient input data for that block is available. The required amount of data is known from the block's current data rate. The scheduler is flexible in nature, allowing changes to each block's data rates during simulation time, thus enabling the simulation of not only SDF models (where all the sample rates are static), but also of DDF models (where data rates change during simulation).

Finally, the VP component contains a bus interface, responsible for communications between the VP component and the processor core(s) in the system over the bus. This block is shown in gray in Figure 3.5, because it needs to be created manually, depending on the bus type, communications protocol and processor core(s) used in the system. However, this manual effort needs to be invested only once, and as long as the platform model does not change, the manually written bus interface can be reused without modification.

The second phase of automatic VP generation is performed by the Virtual Prototype Generator (VPG) tool. This tool extracts all necessary structural information for the particular component from the SSD and creates the interconnected block structure accordingly. Relevant functional information in the SSD is code-styled to be compliant

Component	Structural	Functional	Total
DPE	8	25	33
SYNC	17	39	56
DUD	12	43	55

Table 3.1: *Design effort for manual VP creation*

with the VSIA standard [33] and the C++ language and is then integrated into the VP component. Following these steps, the automatically created VP component can be manually customised to a particular system bus, processor core(s) and communications protocols, before being used.

3.3.3 Results

The environment for automatic generation of VPs presented here has been applied to an industrial design flow of a UMTS receiver in order to estimate its performance benefits. To this end, performance of the automatic VP generation has been compared to that of the usual manual creation of VPs. Both processes start from a completed algorithmic model in Cossap and result in a fully functional VP in complete simulation with bus and CPU models.

As mentioned in Section 3.3.1, Cossap descriptions contain separate structural (interconnection) and functional information. Hence, the design effort (measured in person-hours) for VP creation is composed of processing the structural part of the description, processing the functional part of the description and creation of the interface to the system bus. The first two steps can be performed manually or automatically, whereas the interface to the system bus has to be created manually in both approaches. Hence, the design effort for the interface to the system bus has not been taken into account in this comparison.

Design effort for the manual approach is shown in Table 3.1, with all values in person-hours. The three components for which the results are presented, Delay Profile Estimator

(DPE), Synchronization (SYNC) and Decoding of User Data (DUD), are all parts of the complete, industrially developed UMTS receiver. As can be seen from these results, design effort varied between components, due to their various complexities and code lengths.

Automatic generation of VP components for each of these system blocks took a negligible amount of time (in the order of several seconds). Hence, automatic VP generation produced savings equal to the values shown in Table 3.1, in total exceeding 140 person hours for these three components.

However, with each new revision of the system description at the algorithmic level, the VP generation process has to be repeated. In this industrial UMTS receiver design flow, some system components have well in excess of 50 revisions. Hence, taking into consideration the revision cycles, total savings achieved by the automatic VP generation environment are expected to reach thousands of person-hours.

3.4 Automated Verification Pattern Refinement

As stated previously, design flows for embedded systems traditionally start from initial concepts of system functionality, progressing through a number of refinement steps, eventually resulting in the final product, containing all the software and hardware components that make up the system. These refinement levels of a particular design flow may include the algorithmic level, architectural level, register transfer level (RTL), and others. This is illustrated in Figure 3.6.

The model of the design progresses from one refinement level to the next through the process of **model refinement** (Figure 3.6-A). At each new refinement level, the system model needs to be verified for correct functionality and hence it has associated with it a corresponding set of verification patterns (Figure 3.6-B). The verification patterns at

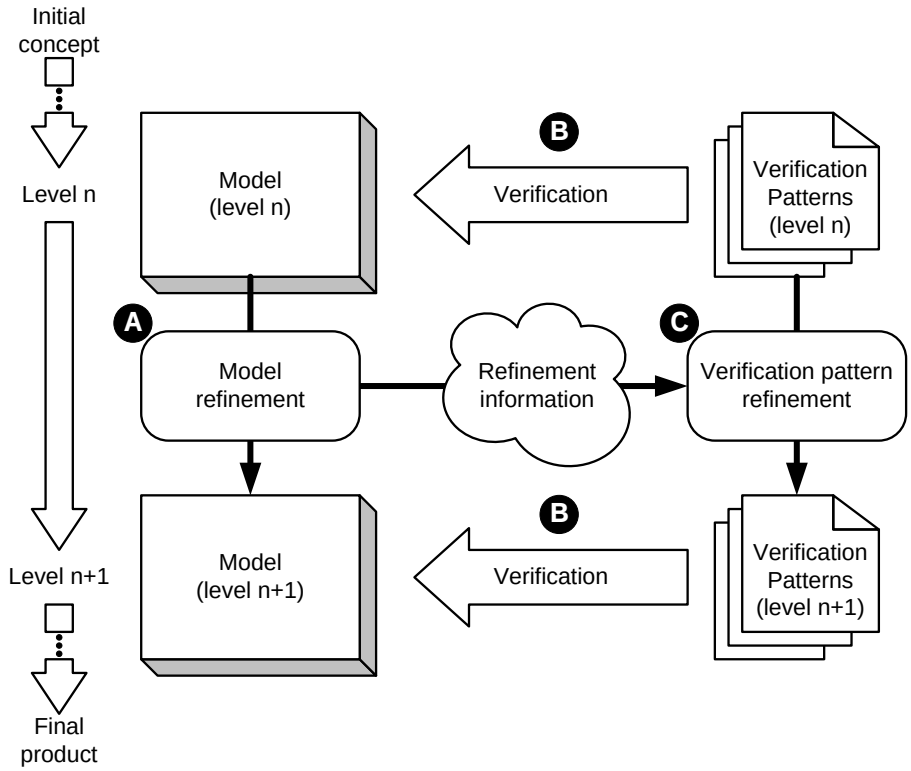

Figure 3.6: *Conceptual view of parallel refinement of the model and the associated verification patterns*

each new level in the design flow are traditionally created from the verification patterns at the previous refinement level (Figure 3.6-C). We refer to this process henceforth as **verification pattern refinement**.

Whereas much research effort is invested in automating model refinement between various refinement levels, verification pattern refinement is in the overwhelming majority of cases still performed manually. This causes both significantly prolonged verification cycles as well as lower design quality, due to the introduction of manual coding errors. Hence, significant reduction of the time to market as well as improvement in quality can be achieved by automating verification pattern refinement.

The manual process of verification pattern refinement, as it is customary in modern engineering practice, involves rewriting of the verification patterns from the earlier refinement

level, applying the refinement information which resulted from model refinement, to produce the new verification patterns (see Figure 3.6). Both sets of verification patterns can be composed of any number of parts, in a variety of languages and modes of interfacing to the model during functional simulation.

The environment for automated generation of virtual prototypes from algorithmic-level models presented in Section 3.3 demonstrated automated model refinement between these two refinement levels. This section presents an environment for automating the corresponding verification pattern refinement, from the algorithmic level to the virtual prototype level.

3.4.1 Verification at Algorithmic Level

At the algorithmic level, the model of the system contains no architectural information and the partitioning of the system is performed on a purely functional basis. Hence, the model of the system typically assumes the form of a process network, with all functional blocks that make up the system executing concurrently and communicating through FIFO channels. Popular commercially available environments for development and simulation of such models are Matlab/Simulink, Cossap, and SPW, among others. The work described here concentrates on algorithmic models developed in the Cossap environment, though with no substantial changes, it is applicable to other algorithmic-level models as well.

The presence of two types of information flowing through the FIFO communications channels of the model is assumed. The first type of information consists of **parameters**, responsible for controlling the modes of operation of each process. The second type of information is **data**, the actual values which are processed in the system and have no influence on the mode of operation of any process.

Therefore, verification patterns at the algorithmic level consist of a set of sequences of values, or **streams**. Exactly one stream exists for each of the data channels going into the model and one for each data channel going out of the model. A pair of dedicated parameter streams, exactly one for all parameters going into the model, and exactly one for those going out of the model, also exist. The complete set of streams is shown as algorithmic-level verification patterns in Figure 3.7.

Since no architectural or implementation information is yet defined at the algorithmic level, the simulation of the model (and hence its verification) at this level is purely untimed functional. In other words, the simulation is driven solely by the availability of input parameters and data, and their processing by the system modules.

3.4.2 Verification at Virtual Prototype Level

As the development of the system progresses, the VP becomes increasingly heterogeneous. Initially, all of the components in the system have a general, purely algorithmic description. During parallel software and hardware development of the various system components (see Figure 3.2), some of the initial component descriptions may be replaced by implementation specific descriptions. For hardware components these may be VHDL or Verilog descriptions, while for software components these may be written in Java or C++ for example.

In this work, we focus on describing verification of system components assigned to hardware implementation, since they will be implemented as part of an HA block (see Figure 3.3). Verification of software components is entirely analogous, but has reduced complexity, because no HA blocks are involved (a more homogeneous problem).

Hence, verification at the virtual prototype level requires the following:

- Device Under Verification (DUV)

- Verification patterns

- Verification program (runs on the DSP, applies the verification patterns to the DUV)

It is important to note that the structure of the hardware platform (see Figure 3.3) enforces the separation of verification patterns into two types, according to how they are communicated to the DUV. Hence, there exist verification patterns communicated to the VP through the system bus (stored in a structured memory image) and those communicated to the VP through its direct I/O interfaces (supplied directly to the VP during functional simulation). Both of these types of verification patterns are shown as virtual prototype level verification patterns in Figure 3.7, together with the necessary verification program.

Since verification at the virtual prototype level relies heavily on transactions over the system bus, it is implemented in a bus-cycle true manner. The bus interface of the DUV, as well as the rest of the simulation environment, including the VSIA-compliant models of the DSP and the system bus, are also accurate to this time resolution within the functional simulation of the complete system.

3.4.3 Environment for Automatic Generation of Verification Patterns

The environment for automated verification pattern refinement presented here [8, 39] generates virtual prototype level verification patterns from algorithmic level verification patterns, as shown in Figure 3.7.

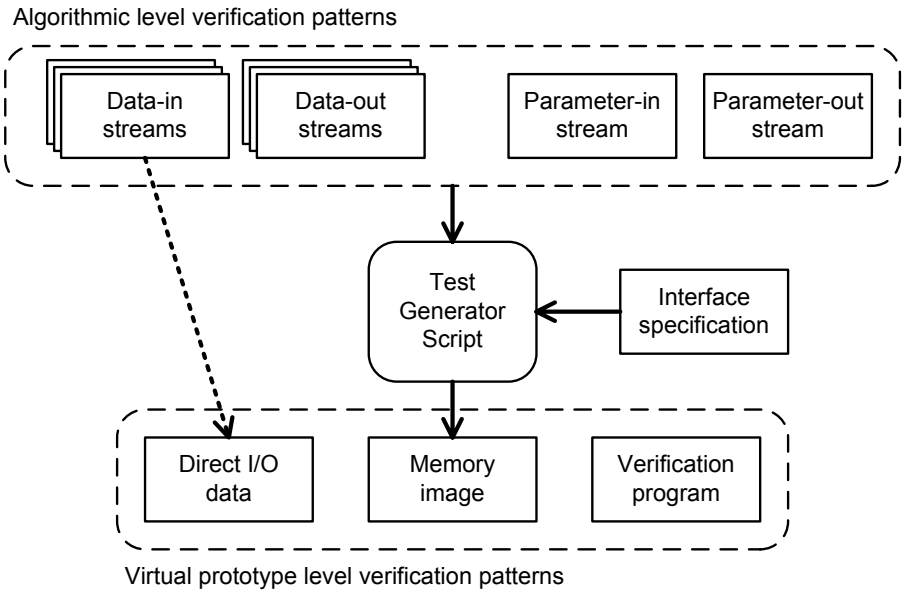

Figure 3.7: *Structure of the environment for automatic generation of verification patterns*

Cossap Verification Patterns

The environment for algorithmic level modeling considered in this work is Cossap from Synopsys. Hence, the algorithmic level verification patterns used also come from the Cossap environment. They are divided into four sets of streams - parameter in and out, and data in and out streams, as shown in Figure 3.7. Exactly one stream exists for all parameters supplied to the DUV during functional verification, as well as exactly one stream for all parameters read from the DUV. Exactly one stream exists for each data input port of the DUV and exactly one for each of its output ports.

The structure of each stream is a simple sequence of values to be supplied to the inputs or expected at the outputs of the DUV. Remembering that verification at the algorithmic level follows an untimed functional paradigm, i.e. is driven purely by the availability of input parameters and data, no further timing information needs to be contained in the streams.

Verification Program

The verification program runs on the processor core and communicates with the DUV over the system bus. Its function is to supply the appropriate verification patterns from the memory image to the DUV, as well as to verify the processing results of the DUV against the expected results, also stored in the memory image. The cycle of writing to/reading from the DUV is repeated for the complete set of verification patterns, on the basis of one input block and one output block being processed per cycle.

Functionality of the verification program is hence not dependent on the particular VP being tested. Thus, the verification program is generic in nature, and can be reused for verification of any VP component. However, a separate verification program must of course be written for every new processor core which is used to run the verification of any DUV.

Memory Image

The memory image is a structured representation of the verification patterns for the virtual prototype level. It includes only those verification patterns which are to be supplied to or read from the DUV over the system bus.

As already mentioned, since the verification program is generic and applicable to the verification of any VP component, all verification pattern values, their sequence and the appropriate interface information must be contained in the memory image. This in turn dictates the structure of the memory image: it contains all the above information, while both making it efficiently accessible in a generic manner by the verification program, as well as minimizing the memory size overhead required to establish this structure.

As a consequence, the memory image is organized as shown in Figure 3.8. It is primarily

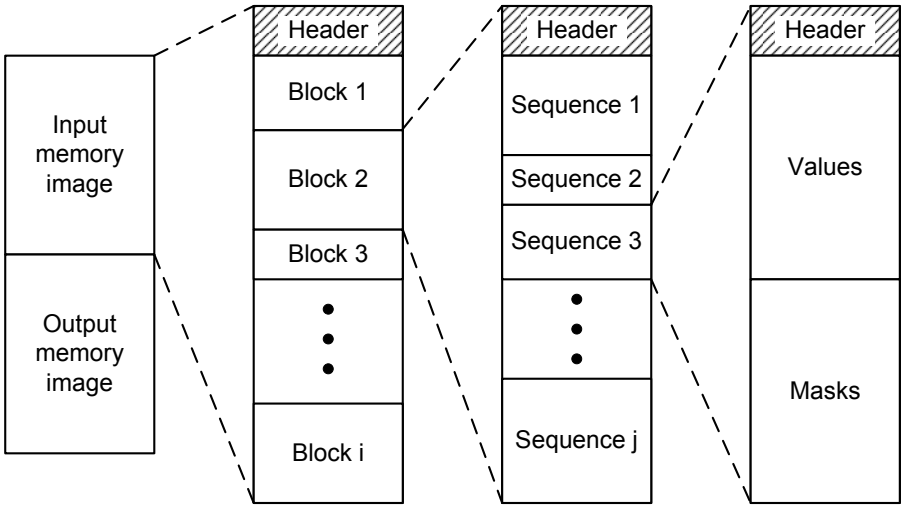

Figure 3.8: *Structure of the memory image*

divided into the **input memory image** and the **output memory image**. The former contains all verification patterns (both parameter and data) which are written to the DUV. The latter contains those verification patterns which are used to check the validity of the outputs of the DUV.

Further, each of the two primary parts of the memory image contains a header, followed by several **blocks**. The header contains the number of blocks in the particular image, followed by a pointer to the beginning of each block, as well as a pointer to the end address of the last block. The latter pointer is effectively the pointer to the end of the particular image and is used in assessing the total size of the memory image by the verification program.

Each block is a set of verification patterns which are consumed (for input image) or produced (for the output image) by the DUV in a single functional invocation. Similar to the structure of the memory image itself, each block contains a header, followed by a number of **sequences**. The header contains the number of sequences in the particular block, followed by a pointer to the beginning of each sequence.

A sequence is a set of verification pattern values to be written to or read from a contiguous section of the DUV's register space. It is composed of a header, a set of **values** and a set of **masks**. The header contains only the start address within the DUV's register space where the write or read operation is to take place.

In the case of the input memory image, the values in a sequence are to be written to the DUV, while the masks determine which bits of each value are to be written to the DUV (overwriting the current content) and which bits are to be kept at their current state. Hence, the required operation for writing the verification patterns from the memory image to the DUV is given (on the bit level) in Equation (3.1). This is a 1-bit multiplex operation, where v is the value in the verification pattern, m is the mask, c is the current value in the DUV register space, and n is the new value.

$$n = (\bar{m} \cdot c) + (m \cdot v) \tag{3.1}$$

In the case of the output memory image, the values in a sequence are to be compared to those returned by the DUV, to verify its functionality. The mask values are used to indicate which of the bits are to be verified and which bits can be regarded as "don't care". Hence, the required operation while verifying the functionality of the DUV is given (on the bit level) in Equation (3.2), where v is the expected value, m is the mask, c is the current value in the DUV register space and t is the test output. A failed test is indicated with the logical state "1" of the variable t.

$$t = m \cdot (c \oplus v) \tag{3.2}$$

Direct I/O Data

As already mentioned in Section 3.4, during the verification process, some verification patterns are supplied to the DUV directly through the I/O interfaces of the HA (see

Figure 3.3) and not through the system bus. Hence, during the verification process these values are not handled by the processor core and are thus not part of the memory image.

The direct I/O data is therefore handled separately during the simulation process. A dedicated module in the simulation environment has been created to serve the sole purpose of making the direct I/O data available to the DUV through its direct I/O ports.

Interface Specification

The interface specification (see Figure 3.7) contains all the structural information which is present, and naturally required during verification, at the VP level, but did not exist at the algorithmic level. Indeed, this interface information comes as a result of the refinement process, going from the algorithmic model to the VP.

In other words, the interface specification is the **refinement information** (as depicted in Figure 3.6) between the algorithmic level and the VP level. Hence, the interface information is needed in order to perform verification pattern refinement between these two levels.

The interface specification can contain interface information for several VP components. Each part dedicated to a particular VP component is composed of exactly one parameter and one data section. The parameter section contains interface information for all the parameters of the VP component in question. Correspondingly, the data section contains interface specifications for each data channel (input as well as output) of the VP component in question.

The parameter interface information includes names of all parameters in the model, to-

gether with their bit-exact addresses in the register table of the DUV. Unlike parameters, data is packaged for communication over the system bus and writing into the register space of the DUV. That is to say, several data values may be packaged into one register of the DUV. If the latter is 32 bits wide, it is efficient to package four 8-bit data values into a single register. Hence, the data section of the interface specification contains in addition to the name of the data input or output, also its packaging factor (being four in the example above) and its starting address in the register space of the DUV.

Test Generator Script

The Test Generator Script (TGS) lies at the core of the automated environment for verification pattern refinement presented here, as shown in Figure 3.7. Its main function is to reformat the algorithmic level verification patterns (in the form of streams) into VP level verification patterns (in the form of the memory image). This script is implemented in the Perl scripting language.

In order to achieve this, the TGS creates the structure of the memory image as described earlier, appropriately formatted for the available algorithmic level verification pattern values. The memory image structure is then filled with both parameter and data verification pattern values, augmented with the architectural information found in the interface specification. The so-prepared memory image is written by the TGS in binary file format, ready to be loaded directly into system memory, either within the VP simulation environment or (in the implementation stage of the design process) on the hardware platform itself.

3.4.4 Example Design

An example design, showing the automated refinement of verification patterns for a virtual component vc1, from the algorithmic level to the virtual prototype level, is given in this section. Initially, this component undergoes refinement of the model itself, as shown in Figure 3.9. Here the model of vc1 in the algorithmic modeling environment,

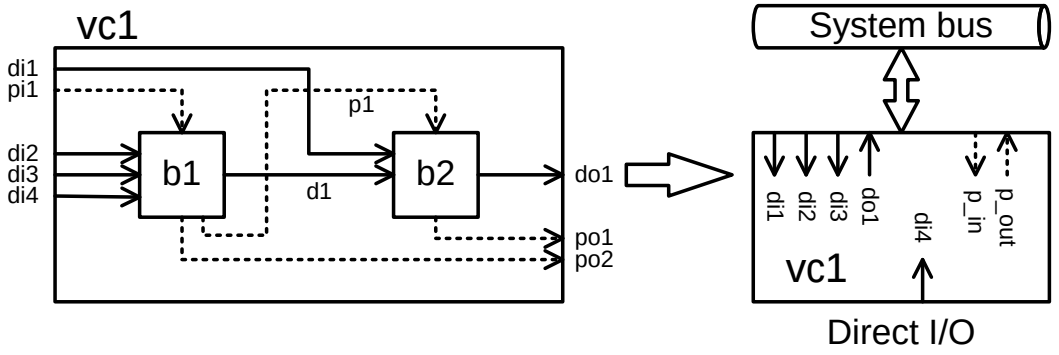

Figure 3.9: *Model refinement of the virtual component vc1, from algorithmic level (left) to virtual prototype level (right)*

such as Cossap, is shown on the left. The component is made up of two sub-blocks, b1 and b2, connected by various data channels (represented by full lines, such as d1) and parameter channels (represented by broken lines, such as p1).

On the right in Figure 3.9, the virtual prototype model of vc1 is shown. This model contains the same interconnected structure as that in the algorithmic model, but *additionally* it contains architectural information. This additional architectural information is hence introduced into the model as a result of the refinement process, shown as **refinement information** in Figure 3.6. This architectural information includes the actual location of data ports, such as the assignment of input port di1 to the system bus interface and input port di4 to the direct I/O interface.

Moreover, this refinement information includes the register mapping of all data and parameter channels which have been assigned to the system bus interface, as described

earlier in this section. The register mapping for the virtual component vc1 is shown in Figure 3.10. Hence, the bus interface between the component vc1 and the processor core

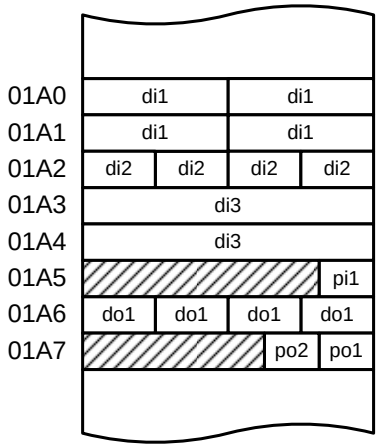

Figure 3.10: *Register mapping of each data and parameter port of vc1*

on which the software components are running occupies the section of the register space between addresses 01A0 and 01A7 (inclusive). Data corresponding to the input data port di1 occupies registers 01A0 and 01A1, with a packaging factor two (as described earlier). Similarly, the output parameters po1 and po2 occupy non-overlapping (but bordering) sections of the register 01A7.

All parts of this refinement information are formally described in the interface specification for the component vc1, as shown in Figure 3.11. Here, it is specified that the input parameter pi1 will be read by vc1 from the address 01A5, occupying a total of four bits, between bits 0 and 3 inclusive. Similar specifications are given for the other parameters. The interface of each data channel is similarly described. For example, data associated with the output data channel do1 is to be written by the component vc1 to the system bus interface, at address 01A6, packaging four data values into each 32-bit register.

After the refinement information has been formally specified, in the form of the interface specification, it is possible to automatically generate virtual prototype verification patterns from algorithmic level verification patterns. These algorithmic level patterns

Interface specification

```
          . . .
COMPONENT vc1
PARAMETER
 pi1 01A5 3 0
 po1 01A7 0 0
 po2 01A7 8 1
DATA
 di1 BUS 01A0 2
 di2 BUS 01A2 4
 di3 BUS 01A3 1
 di4 IO
 do1 BUS 01A6 4

COMPONENT vc2
          . . .
```

Figure 3.11: *Interface specification for the virtual component vc1*

are shown in Figure 3.12. As described earlier, each data input and data output port in

di1	di2	di3	do1	para_in	para_out
NEW BLOCK	NEW BLOCK	NEW BLOCK	NEW BLOCK	NEW BLOCK	NEW BLOCK
CD9A	1B	000B0855	22	pi1 A	po1 0
501C	89	002C4002	01	NEW BLOCK	po2 04
E0D5	60	NEW BLOCK	84	NEW BLOCK	NEW BLOCK
4F05	A1	00F4128E	74	pi1 3	po2 03
NEW BLOCK	NEW BLOCK	00C11032	NEW BLOCK	NEW BLOCK	NEW BLOCK
1AC1	7B	. . .	01	. . .	NEW BLOCK
7000	70		76		po2 1A
. . .	. . .		. . .		. . .

Figure 3.12: *The Cossap verification patterns for each port of vc1*

the algorithmic model has associated with it a stream of values, in addition to the two dedicated parameter streams, para_in and para_out, for the input and output parameters respectively. Values in each stream are divided into blocks, for synchronization across streams.

As already explained, the idea of automated verification pattern refinement revolves around the *enrichment* of the algorithmic level patterns with the refinement information that results from the model refinement, to create virtual prototype patterns automatically. The result is a memory image, containing the original algorithmic patterns, which

are not only reformatted to fit the VP simulation environment (as well as the final hardware platform), but also appropriately enriched with the necessary architectural information, which is not present in the original verification patterns. The structure and content of the memory image for the example virtual component vc1 is shown in Figure 3.13.

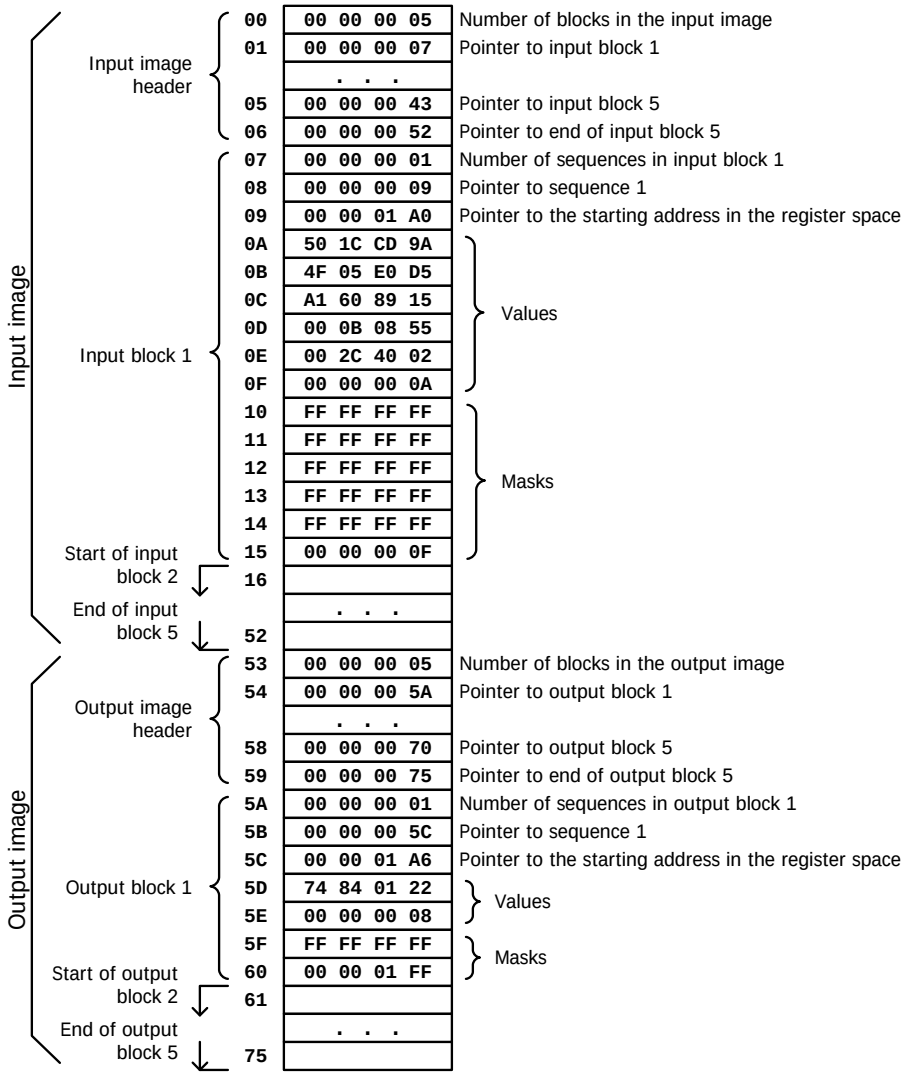

Figure 3.13: *The structure and content of the memory image for the virtual component vc1*

It can be noted that, as explained earlier, the memory image is composed of two parts:

the input and the output image. Each image is then further broken down into a header, followed by a number of blocks. In this case, both images contain five blocks. Each block is composed of a header, followed by a number of sequences. In this example, both the first blocks of the input and the output image are shown fully, and both of them contain one sequence each.

Each sequence starts with a pointer to the starting address in the register space, where the reading (in the case of the input image) or writing (output image) is to start. Following this pointer, the rest of the sequence is made up of actual values and the corresponding masks, as described earlier. In this example, as can be seen in Figure 3.13, the first sequence of the first block of the input image is six values long, whereas the same in the output image is two values long.

3.5 Conclusions

In addition to the inherent acceleration of the design process through virtual prototyping, the virtual prototyping environment integrated into OTIE has also been shown to produce significant further design effort savings, in the order of hundreds to thousands of person-hours [7].

The extension of this virtual prototyping environment into the verification flow has been shown to fully automate the transition of the design from the algorithmic to the virtual prototype level [8, 39]. This is a unique ability of the OTIE, which effectively serves as its proof of concept. In other words, the fully automated virtual prototyping environment clearly demonstrates the capability of the OTIE to integrate and support a complete, industrial design flow, made up of both industrial and in-house developed tools, both in the model refinement and verification flows.

4. AUTOMATED FLOATING-POINT TO FIXED-POINT CONVERSION WITH THE *FIXIFY* ENVIRONMENT

Design of embedded systems typically starts with the initial concept of the system, which is at the so-called algorithmic level turned into an executable model. The high-level specifications of the system can be verified on this algorithmic model. Development at this stage of the design process is aided by such tools as Matlab/Simulink, CoWare SPW, Synopsys CoCentric System Studio or Ptolemy [40].

In terms of the numeric formats used, algorithmic-level models invariably use floating-point formats, for several reasons. Firstly, while the algorithm itself is undergoing changes, it is necessary to disburden the designer from having to take numeric effects into consideration. Hence, using floating-point formats, the designer is free to modify the algorithm itself, without taking into consideration overflow and quantization effects. Also, floating-point formats are highly suitable for algorithmic modeling because they are natively supported on PC or workstation platforms, where algorithmic modeling usually takes place.

On the other hand, at the end of the design process lies the implementation stage, where all hardware and software components of the system are fully implemented in the chosen target technologies and then integrated to create the final product. At this design stage, various compilers, debuggers, and place-and-route tools are the EDA tools most commonly used. Both the software and hardware components of the system at this stage use only fixed-point numeric formats, because the use of fixed-point formats

allows drastic savings in all traditional cost metrics: the required silicon area, power consumption and latency/throughput (i.e. performance) of the final implementation.

Thus, during the design process it is necessary to perform the conversion from floating-point to suitable fixed-point numeric formats, for all the data channels in the system. Please refer to Section 2.5.2 for the definition of data channels and their role in the system description. This transition necessitates careful consideration of the ranges and precision required for each channel, the overflow and quantization effects created by the introduction of the fixed-point formats, and possible instability effects these may introduce.

A trade-off optimization is hence formed, between minimizing the implementation cost of the system, while at the same time maintaining the numeric performance of the system above a specified requirement. The system implementation cost is directly related to the overall bitwidths of all the signals in the system. The numeric performance of the system is typically measured in terms of the Signal-to-Quantization Noise Ratio (SQNR) [41, 42, 37]. The SQNR is defined as the logarithmic ratio of the quantization noise introduced by the use of fixed-point formats to the value of the signal itself. Please see Section 4.2.4 for a formal definition of the SQNR.

This performance-cost trade-off is traditionally performed manually, with the designer estimating the effects of fixed-point formats through system simulation and determining the required bitwidths and rounding/overflow modes through previous experience or given knowledge of the system architecture (such as predetermined bus or memory interface bitwidths). This iterative procedure is very time-consuming and can sometimes account for up to 50% of the total design effort [41]. Hence, a number of approaches to automating the conversion from floating-point to fixed-point formats have been proposed.

4.1 Related Work

Existing approaches to automating the floating-point to fixed-point conversion can be clearly divided into the analytical (or static) and statistical (or dynamic) approaches. Each group has a number of advantages and limitations, as described below.

4.1.1 Analytical Approaches

All the analytical approaches to automating the conversion from floating-point to fixed-point numeric formats find their roots in the static analysis of the algorithm in question. The algorithm, represented as a Control and Data Flow Graph (CDFG), is statically analyzed, propagating the bitwidth requirements through the graph, until the range, precision, and sign mode requirements of each signal are determined.

As such, analytical approaches do not require any simulations of the system to perform the conversion. This typically results in significantly improved runtime performance, which is the main benefit of employing such a scheme. Also, analytical approaches do not make use of any input data for the system. This relieves the designer form having to provide any data sets with the original floating-point model and makes the results of the optimization dependent only on the algorithm itself and completely independent of any data which may eventually be used in the system.

However, analytical approaches suffer from a number of critical drawbacks in the general case. Firstly, analytical approaches are inherently only suitable for finding the upper bound on the required precision, and are unable to perform the essential trade-off between system performance and implementation cost. Hence, the results of analytical optimizations are excessively conservative, and cannot be used to replace the designer's fine manual control over the trade-off. While modifications to the basic principle do

allow building-in of controls through which the optimizations results are offset from the conservative upper bound, these controls are very coarse and not generally suitable for carrying out the performance/cost trade-off.

Furthermore, analytical approaches are not suitable for use on all classes of algorithms. Since the static analysis of the algorithm typically reduces to computing the transfer function between its inputs and outputs, for the purpose of propagating the bitwidth requirements through the CDFG representation, it is in general not possible to process non-linear, time-varying, or recursive systems with these approaches.

Keding et al. [41] presented FRIDGE, one of the earliest environments for floating-point to fixed point conversion, based on an analytical approach. This environment has high runtime performance, due to its analytical nature, and wide applicability, due to the presence of various back-end extensions to the core engine, including the VHDL back-end (for hardware component synthesis) and ANSI-C and Assembly back-ends (for DSP software components). However, the core engine relies fully on the designer to pre-assign fixed-point formats to a sufficient portion of the signals, so that the optimization engine may propagate these to the rest of the CDFG structure of the algorithm. This environment is based on fixed-C, a proprietary extension to the ANSI-C core language and is hence not directly compatible with standard design flows. The FRIDGE environment forms the basis of the commercial Synopsys CoCentric Fixed-Point Designer [43] tool.

Another analytical approach, Bitwise, was presented by Stephenson et al. [44]. Bitwise implements both forward and backward propagation of bitwidth requirements through the graph representation of the system, thus making more efficient use of the available range and precision information. Furthermore, this environment is capable of tackling complex loop structures in the algorithm by calculating their closed-form solutions and using these to propagate the range and precision requirements. However, this environ-

ment, like all analytical approaches, is not capable of carrying out the performance-cost trade-off and results in very conservative fixed-point formats.

An environment for automated floating-point to fixed-point conversion for DSP code generation, presented by Menard et al. [45], minimizes the execution time of DSP code through the reduction of variable bitwidths. However, this approach is only suitable for software components and disregards the level of introduced quantization noise as a system-level performance metric in the trade-off.

An analytical approach based on affine arithmetic was presented by Fang et al. [46]. This is another fast, but conservative environment for automated floating-point to fixed-point conversion. The unique feature of this approach is the use of a *probabilistic relaxation* scheme, in order to combat the typically pessimistic nature of the analytical optimization results. Rather than propagating actual true intervals of system variables through a graph representation of the system, the authors instead propagate a confidence interval which bounds the true interval with a specified probability. The authors introduce an adjustable parameter λ to represent this probability. In a normal hard upper bound analysis, λ equals to exactly 1, that is to say, the confidence interval always covers the entire true interval of the variable. Through this novel probabilistic relaxation scheme, the authors are able to set $\lambda = 0,999999$ and thereby achieve significantly more realistic optimization results, that is to say, closer to those achievable by the designer through system simulations.

However, while this scheme provides a method of relaxing the conservative nature of its core analytical approach, the mechanism of controlling this separation (namely the trial-and-error search by varying the λ factor) does not provide a means of controlling the performance-cost trade-off itself and thus replacing the designer. Also, though this approach does use the standard SystemC fixed-point library for bit-true simulations, it is

still based on a custom extension to the C++ language for the analytical propagation and is capable of analyzing only addition, subtraction, multiplication, and division operators.

4.1.2 Statistical Approaches

The statistical approaches to performing the conversion from floating-point to fixed-point numeric formats are based on performing system simulations and using the resulting information to carry out the performance-cost trade-off, much like the designer does during the manual conversion.

Due to the fact that these methods employ system simulations, they may require extended runtimes, especially in the presence of complex systems and large volumes of input data. Hence, care has to be taken in the design of these optimization schemes to limit the number of required system simulations. Also, statistical approaches obligate the designer to provide input data together with the original floating-point design. However, such input data is in the vast majority of the cases immediately available, since the design at the algorithmic level (before reaching the floating-point to fixed-point conversion) needs to be thoroughly verified.

The advantages of employing a statistical approach to automate the floating-point to fixed-point conversion are numerous. Most importantly, statistical algorithms are inherently capable of carrying out the performance-cost trade-off, seamlessly replacing the designer in this design step. Also, all classes of algorithms can be optimized using statistical approaches, including non-linear, time-varying, or recursive systems.

One of the earliest research efforts to implement a statistical floating-point to fixed-point conversion scheme was presented by Kim et al. [42],concentrating on DSP designs represented in C/C++. This approach shows high flexibility, characteristic to statistical approaches, being applicable to non-linear, recursive, and time-varying systems.

However, while this environment is able to explore the performance-cost trade-off, it requires manual intervention by the designer to do so. The authors employ two optimization algorithms to perform the trade-off: full search and a heuristic with linear complexity. The high complexity of the full search optimization is reduced by grouping signals into clusters, and assigning the same fixed-point format to all the signals in one cluster. While this can reduce the search space significantly, it is an unrealistic assumption, especially for custom hardware implementations, where all signals in the system have very different optimal fixed-point formats. This environment is based on a custom extension of the C++ language, thus making the design environment incompatible with other EDA tools.

A hybrid analytical-statistical environment for floating-point to fixed-point conversion was presented by Cmar et al. [47]. This approach uses analytical means to discover the required ranges (and hence determine the MSB requirements), while employing a statistical scheme to discover the precision requirements (i.e. LSB requirements) of all the data channels in the system. A unique feature of this approach is the use of an empirical constant K_{em}, which is used as an indirect control of the acceptable SQNR, thus carrying out the performance-cost trade-off.

However, this environment does not provide a fully automated optimization algorithm to effect the performance-cost trade-off. Rather, the designer drives the environment manually to achieve the desired trade-off point. Also, this environment is based on custom extensions of the C++ language, thus making it not directly compatible with other EDA tools.

Cao et al. [48] presented QDDV, an environment for floating-point to fixed-point conversion aimed specifically at video applications. The unique feature of this approach is the use of two performance metrics. In addition to the widely-used *objective* metric, the

SQNR, the authors also use a *subjective* metric, the Mean Opinion Score (MOS) taken from ten observers.

While this environment does employ a statistical framework for measuring the cost and performance of a given fixed-point format, no automation is implemented and no optimization algorithms are presented. Rather, the environment is available as a tool for the designer to perform manual "tuning" of the fixed-point formats to achieve acceptable subjective and objective performance of the video processing algorithm in question. Additionally, this environment is based on Valen-C, a custom extension to the ANSI-C language, thus making it incompatible with other EDA tools.

A further environment for floating-point to fixed-point conversion based on a statistical approach was presented by Shi et al. [49]. This environment is aimed at optimizing models in the MathWorks Simulink [50] environment. This approach derives an optimization framework for the performance-cost trade-off, but provides no optimization algorithms to actually carry out the trade-off, thus leaving the conversion to be performed by the designer manually.

Here we present *fixify*, an environment for automated floating-point to fixed-point conversion, based on a statistical approach and implemented as an integrated part of the OTIE. The *fixify* environment supersedes all the previously presented environments through its ability to completely replace the designer and fully automate the conversion process. Also, it is based on SystemC (although its modular structure also allows use of other modeling languages as well), and is thus ready to be seamlessly integrated into all development flows using this language. The *fixify* environment also offers a range of optimization methods for performing the conversion, allowing the designer to automatically exercise the finest possible control over the cost-performance trade-off for any implementation scenario.

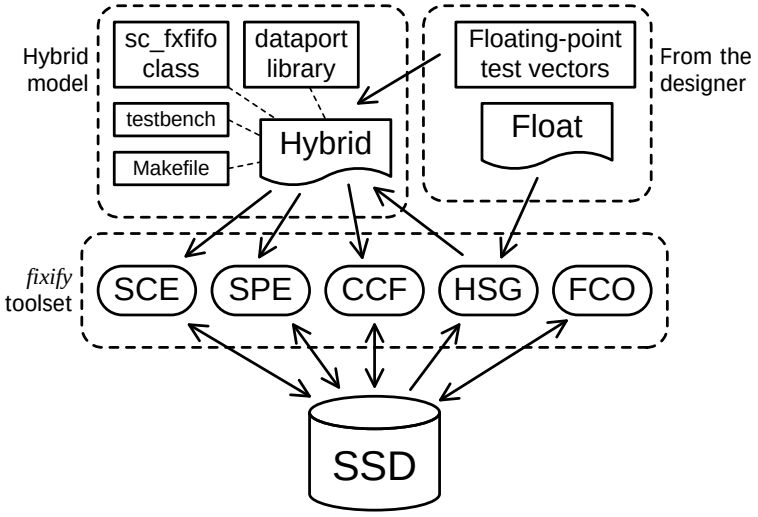

Figure 4.1: *Structure of the fixify environment*

The rest of this chapter is organized as follows. Section 4.2 describes the structure and organization of the *fixify* environment, followed by Section 4.3, describing the various optimization methods it offers to the designer. Results of processing a typical industrial DSP design with the *fixify* environment are presented in Section 4.4, including comparison of the optimization methods and the achievable control over the performance/cost trade-off. Finally, the conclusions are drawn in Section 4.5.

4.2 The *fixify* Environment

The *fixify* environment is a tool chain developed within the OTIE to provide support for automated floating-point to fixed-point conversion. Hence, it is built around the core of the OTIE, the SSD, and employs a statistical approach to the optimization of fixed-point formats of all data channels in the system.

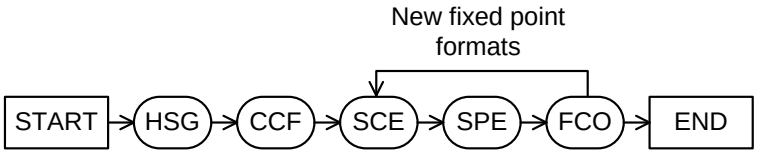

Figure 4.2: *Typical work flow through the fixify tool chain*

4.2.1 Structure of the Environment

A set of five tools forms the *fixify* environment, as shown in Figure 4.1: the Hybrid System Generator (HSG), the Corner Case Finder (CCF), the System Performance Estimator (SPE), the System Cost Estimator (SCE), and the Fixed-point Configuration Optimizer (FCO).

The initial input by the designer is the pure floating-point design and the set of floating-point test vectors. The hybrid model is generated by the HSG tool and used by the CCF tool to discover the corner-case fixed-point configuration of the system. The SCE tool is used to estimate the implementation cost of the final system and the SPE tool uses the hybrid description to estimate the numeric performance of the current fixed-point configuration. Finally, the FCO tool is used to perform the trade-off between system cost and performance and thus optimize the fixed-point formats of all data channels in the system. The work flow as a design is processed by the *fixify* toolset is iterative, as shown in Figure 4.2.

4.2.2 Hybrid Model

The hybrid model of the system is used to estimate the degradation of its numeric performance by the introduction of fixed-point formats, relative to the pure floating-point model. This performance degradation is estimated through simulation of the hybrid model.

In this work, it is assumed that the original SystemC floating-point design, as supplied by the designer, is composed of **functional** and **structural** instances of modules, as is the usual design practice. Functional instances are those which contain one or more concurrent processes, which implement the functionality of the system. Structural instances are those which provide the structure and interconnections that make up the system.

The functional and structural instances are connected by data channels, altogether representing the design in the form of a graph, where instances are the nodes and data channels are the edges. The *fixify* environment optimizes the fixed-point formats of the data channels at this level of representation of the system, i.e. optimizes the formats of the edges of this graph. For systems that contain no feedback data channels, the graph representation of the system is a Directed Acyclic Graph (DAG).

The key characteristic of the hybrid model is its ability to set any of the channels in the design to either floating-point or any fixed-point format. Hence, the entire hybrid system can then be used to simulate any combination of various fixed-point and floating-point formats in the design. It is also important to note that the hybrid model of the *fixify* environment is capable of fully transforming the numeric formats of any number of its channels at run time, i.e. post compilation.

This is achieved through environment variables, which are monitored by the hybrid model and can be set by any of the tools (such as CCF or SPE), thus achieving finest-grain control of all numeric formats in the hybrid model at run time. Aside from the interface convenience this mechanism offers, its most important advantage is the dramatic time savings it offers to the *fixify* environment. Since the hybrid model can be reconfigured at run time, during the optimization process, it needs to be compiled just once, rather than with every change of the fixed-point configuration.

Word length	Arbitrary integer value
Integer word length	Arbitrary integer value
Sign mode	Unsigned
	Signed (two's complement)
Overflow mode	Saturation
	Saturation to zero
	Symmetrical saturation
	Wrap-around
	Sign magnitude wrap-around
Quantization mode	To $+\infty$
	To zero
	To $-\infty$
	To ∞
	Convergent rounding
	Truncation
	Truncation to zero

Table 4.1: *Controls over fixed-point formats available in the SystemC language*

The hybrid model is created by the HSG tool, as shown in Figure 4.1. Since the HSG tool operates directly on the system description provided by the designer, it is language-specific. Currently, the HSG tool supports SystemC [20] descriptions, but is built in a modular fashion, allowing for easy extension to other system description languages as well.

The *fixify* environment hence has access to and makes full use of all the controls over fixed-point formats available in the SystemC language [51]. These controls are summarized in Table 4.1.

Also, SystemC provides two fast fixed-point data types, `sc_fix_fast` and `sc_ufix_fast`, for signed and unsigned values respectively [51]. These datatypes are useful in simulating limited-precision fixed-point values, namely with mantissa lengths up to 53 bits. By exploiting the reduced mantissa lengths of these types, the SystemC reference implementation library is able to achieve significant gains in simulation performance. In the *fixify* environment, these limited precision data-types are used whenever possible, to

achieve maximum simulation performance.

The HSG tool uses unaltered functional instances of the original floating-point design to build the hybrid model. However, it modifies the structural instances to use `sc_fxfifo` channels, developed as part of the *fixify* environment. These channels implement the dynamic control over numeric formats in the system, as described above.

Also generated by the HSG tool is a testbench for the design (see Figure 4.1). The automatically generated testbench contains interfaces to the supplied floating-point test vectors through the use of the `dataport` library, also written as part of the *fixify* environment. Compilation of the whole design is also automated by the HSG tool through an automatically generated "Makefile" for the system. After the hybrid model is compiled, it is delivered as an executable, for use by the rest of the *fixify* tool chain.

4.2.3 Corner Case

Introduction of fixed-point formats into a pure floating-point design creates degraded numeric performance of the system, through the introduction of both quantization and overflow errors. The optimization of fixed-point formats in the system is a trade-off between the reduced performance of the system and the reduced cost of the implementation through the reduction of fixed-point bitwidths.

At the extreme end of this trade-off lies the so-called **corner case fixed-point configuration**, defined as the set of minimum fixed-point formats for the entire system which produces no change to the numeric performance compared to the floating-point system and hence sets the maximum acceptable cost for the system implementation. Thus, the corner case is the ideal starting point for the optimization process, as all other fixed-point configurations which will be considered will have both worse performance, but also lower implementation cost, than the corner case configuration.

It is the sole purpose of the CCF tool to find the corner case configuration for the hybrid system and place this information back into the SSD. The CCF tool first runs the hybrid system with all channels in floating-point format, and analyzes the bitwidth and sign format (signed/unsigned) requirements of each channel. Following this analysis, the CCF tool verifies the corner-case formats it found by running the hybrid system in this configuration and checking that the operation of the system is identical to that with floating-point formats. Once the corner case configuration is verified, the CCF tool stores this design refinement information into the SSD.

4.2.4 Cost and Performance Estimation

The two key metrics that are considered in the process of optimizing fixed-point formats in a design are the numeric performance of the system and its implementation cost. In the *fixify* environment, these metrics are estimated by the SPE and the SCE tools respectively.

Relative degradation of numeric performance of a system through the introduction of fixed-point formats is most commonly measured by the SQNR [41, 42, 37]. Recall that the SQNR is defined as the logarithmic ratio of the quantization noise introduced by the use of fixed-point formats to the value of the signal itself. Hence:

$$\text{SQNR} = 20 \times \log\left(\frac{1}{E}\right) \tag{4.1}$$

In the above expression, E is the relative error, or the relative difference between the original and quantized data values, v_{orig} and v_{quan} respectively.

$$E = \left|\frac{v_{orig} - v_{quan}}{v_{orig}}\right| \tag{4.2}$$

It is important to note that (from Equation (4.1)) the knowledge of the relative error alone is sufficient to determine the SQNR. Hence, in the *fixify* environment, performance is expressed in terms of the relative error, E. This brings the advantage of reduced

E_{max_tot}	Maximal total error
E_{max_ins}	Maximal inserted error
E_{avg_tot}	Average total error
E_{avg_ins}	Average inserted error

Table 4.2: *Error metrics used by the fixify environment*

computational complexity, as well as easier handling of the case of no quantization error, where $E = 0$ and SQNR $= -\infty$.

The SPE tool estimates four performance metrics and places this refinement information back into the SSD. These metrics are shown in Table 4.2. The maximum error metrics, E_{max_tot} and E_{max_ins}, are the peak errors found in each data channel throughout the functional simulation. On the other hand, the average error metrics, E_{avg_tot} and E_{avg_ins}, are the average errors in each data channel throughout the functional simulation.

Furthermore, in all the data channels in the system during the functional simulation, there exist two error components. The first is the **inherited** error, that is to say the error which has not been created by the fixed-point format of the channel itself, but rather by the fixed-point formats of its predecessor data channels. These are the other data channels in the system which are closer to the system inputs. The second error component is the **inserted** error, that is to say the error which has been created by the fixed-point format of the data channel in question.

Hence, the inserted error metrics, E_{max_ins} and E_{avg_ins}, during their calculation take into account only the inserted error component, whereas the total error metrics, E_{max_tot} and E_{avg_tot}, take both error components into consideration.

The designer controls the performance-cost trade-off by setting the lower bound on the system performance, in the form of an SQNR$_{min}$ value for the system. This value

specifies the lowest acceptable level of SQNR for the entire system and corresponds through Equation (4.1) to E_{lim}, the maximum allowable peak error in the system. Throughout the performance-cost trade-off, the optimization algorithms accept only those solutions for which the Inequality 4.3 holds true.

$$E_{max\_tot} \leq E_{lim} \tag{4.3}$$

Implementation cost of the system is also needed to perform the trade-off optimization that determines all the fixed-point formats in the system. The key requirement of the cost metric is to reflect the choice of bitwidth for all data channels in the system on its implementation cost. As such, it is independent of the actual implementation of each system component (hardware or software) and hence needs not be a firm **absolute** measure (such as seconds of run time for software or gate count for hardware). However, it must be a good **relative** measure, so as to enable the optimization engine to compare correctly the implementation costs of any number of fixed-point configurations. Hence, the SCE tool calculates the cost metric c as a sum of all channel bitwidths. This is shown in Equation (4.4), where the number of data channels in the system is given as n and the bitwidth of the i^{th} channel is w_i. The SCE tool stores this refinements information back into the SSD.

$$c = \sum_{i=1}^{n} w_i \tag{4.4}$$

Since neither the SPE nor the SCE tool interact directly with any system description code (neither the original floating-point description nor the hybrid model), both of these tools are independent of the description language used and require no modification in order to be used with languages other than SystemC.

4.2.5 Optimization Engine

The optimization engine of the *fixify* environment is implemented in the FCO tool. Since this tool interfaces only to the SSD (see Figure 4.1), it is also independent of the system description language used and hence requires no modification to be used with languages other than SystemC.

The FCO tool is a general framework for implementing optimization algorithms, providing interfaces to the performance and cost estimates in the SSD, as well as control over the numeric formats of all channels in the design. Hence, it is suitable for implementation of any number of various optimization methods. The optimization methods currently implemented in the FCO tool are described in detail in Section 4.3.

4.3 Optimization Methods

The optimization engine of the *fixify* environment offers to the designer three different optimization algorithms: restricted-set full search, greedy, and branch-and-bound.

4.3.1 Full Search

The basic optimization technique guaranteed to produce optimal results is the full search through the solution space [52]. The most serious drawback of this approach is its long run time, due to the fact that each possible solution needs to be considered separately.

For the floating-point to fixed-point conversion problem, the search space grows extremely quickly. If the number of data channels in the system is given as n and the corner-case bitwidth of the i^{th} channel is given as W_i, the size of the search space, s, is given as:

$$s = \prod_{i=1}^{n} W_i \tag{4.5}$$

In other words, the total number of solutions considered by the full search algorithm grows exponentially with n, the number of channels in the system. This is shown in Equation (4.6), where W_{min} and W_{max} are the minimum and maximum corner-case bitwidths respectively, considered over all the channels in the system.

$$W_{min}^{n} \leq s \leq W_{max}^{n} \tag{4.6}$$

The range of W_i values is typically between 50 and 100, and even small designs contain 5 to 10 channels. Thus, the full search algorithm running on such a design needs to consider approximately $75^8 \approx 10^{15}$ possible solutions. To evaluate all the solutions in this vast search space, even with the simulation time of the hybrid model as short as say 0,5 seconds, the full search algorithm would take in excess of 15 million years to complete! Therefore, applying the full search algorithm to even the smallest of DSP designs is impractical.

4.3.2 Restricted-set Full Search

Although all variations on the basic full search algorithm require consideration of each possible solution in the search space and hence have complexity that grows exponentially with the number of channels in the system, it is possible to reduce the run time of the full search algorithm down to the reasonable range of several minutes to several hours.

This can be achieved by reducing the set of possible bitwidths that can be assigned to any channel in the system from N, the set of all positive integers, to a much smaller set, such as $\{16, 32, 64\}$ for example. Hence, the set of possible bitwidths that can be

applied to the i^{th} channel is made up of the closest larger bitwidth than W_i in the global set of possible bitwidths, and all the bitwidths smaller than that. For example, if W_i is 11 and the global set of possible bitwidths is $\{4, 8, 16, 32\}$, the set of bitwidths which can be applied to the i^{th} channel is $\{4, 8, 16\}$. This ensures that if all the bitwidths smaller than W_i fail to satisfy the performance criteria, i.e. SQNR_{min}, the one bitwidth which exceeds W_i is the fail-safe solution which is guaranteed to provide satisfactory performance, assuming a monotonic performance function with respect to individual channel bitwidths. Seen in another way, only one of the possible bitwidths which are larger than W_i needs to be considered - the smallest one. Thus, given that p_i is the number of these possible bitwidths that can be applied to the i^{th} channel, the search space can now be expressed as

$$s = \prod_{i=1}^{n} p_i \tag{4.7}$$

Again, as can be seen in Equation (4.8), the search space grows exponentially with n, the number of data channels in the system, but with a drastically reduced base (p instead of W).

$$p_{min}^n \leq s \leq p_{max}^n \tag{4.8}$$

By restricting the set of possible bitwidths, the example mentioned earlier reduces in complexity from 10^{15} to just $3^8 = 6561$, which can be computed in less than an hour.

This approach is not only practical, but also highly suitable for optimization of designs that are aimed for implementation on DSP architectures, which will typically offer a restricted set of possible bitwidths, such as for example $\{16, 32, 40\}$ on the TI TMS320C62x architecture. It is also important to note that the restricted-set full search

algorithm is guaranteed to find the optimal set of fixed-point formats for all data channels in the system.

4.3.3 Greedy Search

If the requirement of finding the guaranteed global optimum within the search space is relaxed, it is possible to implement optimization techniques that find near-optimal solutions in much shorter periods of time. One of these techniques is the greedy search algorithm [52]. This algorithm is based on making steps through the search space, and when choosing the direction of the next step, always choosing the locally most favorable direction.

The greedy search algorithm of the *fixify* environment initially sorts the data channels in the design hierarchically, starting with the outputs, systematically working its way through the structure of the design, finishing with its inputs. If any cycles (i.e. feedback channels) exist in the design, these cycles are resolved by considering each channel in the cycle exactly once. Once the optimization sequence of data channels is determined, the bitwidth of each channel is optimized, keeping the formats of the rest of the channels fixed. This "upstream" sequence of optimization ensures minimal interference through inherited quantization error as each channel is optimized.

Therefore, each channel is optimized separately and only once, and the upper bound on the complexity of the greedy algorithm is given by:

$$s = \sum_{i=1}^{n} W_i \tag{4.9}$$

As can be seen in Equation (4.10), the number of required simulations thus grows linearly with n, the number of channels in the design.

$$nW_{min} \leq s \leq nW_{max} \tag{4.10}$$

For the example design discussed earlier, the greedy search optimization has the complexity of only $75 \times 8 = 600$ (down from the original 10^{15}), and thus computes in just 5 minutes.

However, it must be noted that although greedy optimizations typically produce excellent results, they offer no guarantee of finding the optimum fixed-point configuration for the system.

4.3.4 Branch-and-Bound

Another optimization technique which produces faster results than the full search optimization is the branch-and-bound optimization [52]. Branch-and-bound algorithms are based on the idea of representing the search space in the form of a tree structure and minimizing the time required to find the optimal solution by intelligent traversal of this tree.

Indeed, branch-and-bound algorithms systematically exclude sub-branches of the tree relative to their current location, when they can infer that these sub-branches cannot contain the optimum solution. These algorithms thus make overwhelming savings in complexity by being able to exclude parts of the search space through their inherent knowledge of the problem. In other words, branch-and-bound algorithms find the same global optimum full search algorithms do, but contain intelligent and problem-specific mechanisms to dramatically shrink the search space. In particular, the monotonic nature of the performance function with respect to channel bitwidth is exploited.

The branch-and-bound optimization technique of the *fixify* environment starts off by

producing the optimization sequence of data channels, in the same way greedy search optimization does. Data channels are then optimized individually, again in the same way greedy optimization does, but having found the minimal bitwidth of each channel this way, instead of proceeding to the next channel in the sequence, branch-and-bound optimization does not discard the rest of the solutions in the current subtree of the solution space, but proceeds to look through it, looking for the global optimum.

Once the bitwidth of the current channel is minimized, the branch-and-bound algorithm assembles the list of data channels in the design that are influenced by the current channel, i.e. are found "downstream" in the dataflow graph of the system. Following this, all possible combinations of tightening the current channel further and relaxing any combination of the subtree channels is explored.

However, rather than considering every possible combination like the full search algorithm would, the branch-and-bound algorithm takes further shortcuts during this exploration process, i.e. excludes parts of the solution subtree. Any possible solutions which increase the implementation cost are not considered. This removes a very significant part of the search space. Also, the iterative tightening of the current channel and relaxation of the subtree channels does not go on beyond the point where the inserted maximal error (E_{max_ins}) in the current channel exceeds the global threshold (E_{lim}) set by the designer, which also removes a significant part of the search space.

While the worst-case complexity of the branch-and-bound algorithm is equal to that of full search optimization (see Equation (4.5)) and grows exponentially with the number of channels in the system, this optimization typically executes orders of magnitude faster, due to the minimization of the search space. On the other hand, branch-and-bound optimization is under two restrictions guaranteed to find the optimum fixed-point configuration for the system, just like full search optimization.

The first condition necessary to guarantee finding the optimum fixed-point configuration is that the system's dataflow graph can be represented as a DAG, i.e. the system does not include feedback signals. Systems which contain feedback cycles in the dataflow graph cannot be resolved to an optimization sequence where all the signals in the subtree of the current channel do not influence the current channel itself. Such systems can be optimized by the branch-and-bound algorithm, by breaking the cycle in the dataflow graph. The system is then analyzed as described above, but results cannot be guaranteed to be optimal.

The second necessary condition to guarantee finding the optimum fixed-point configuration by employing the branch-and-bound algorithm described here is that the monotonic nature of the system performance function is maintained, that is to say the numeric performance of the system stays the same or improves (but does **not** deteriorate) with increased precision. It is possible to optimize designs for which this property does not hold, but the resulting fixed-point configuration cannot be guaranteed to be optimal.

4.4 Results

In order to demonstrate the viability of employing the *fixify* environment on a realistic DSP design, as well as investigate the relative performance of its three optimization techniques, the results of processing an industrial DSP design by the *fixify* environment are shown here. The design used as a benchmark is a Multiple-Input, Multiple-Output (MIMO) receiver design, taken from a Wireless Local Area Network (WLAN) 802.11n implementation [53].

The MIMO receiver design, described in SystemC, is broken down into two functional modules, connected through eight data channels, as shown in Figure 4.3. The estimation of channel coefficients is implemented in the module `chan_est`, which reads in 212

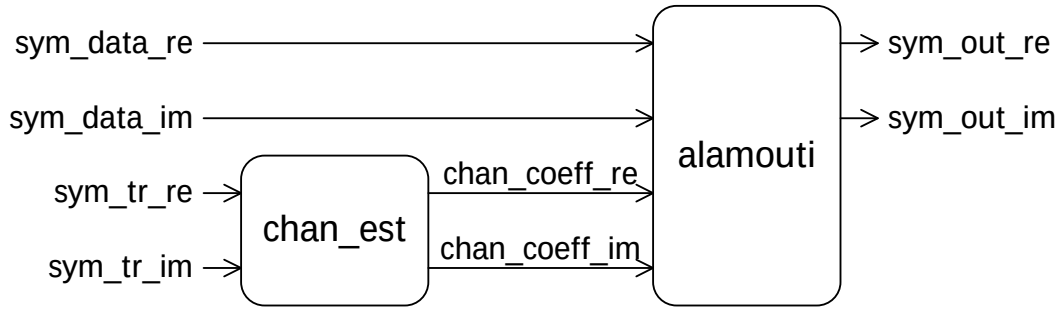

Figure 4.3: *Structure of the MIMO receiver design*

Name	Sign mode	MSB	LSB	Bitwidth
sym_tr_re	signed	13	-2	16
sym_tr_im	signed	13	-2	16
chan_coeff_re	signed	12	5	8
chan_coeff_im	signed	12	5	8
sym_data_re	signed	13	-2	16
sym_data_im	signed	13	-2	16
sym_out_re	signed	25	18	8
sym_out_im	signed	25	18	8

Table 4.3: *An example optimization result: full search(8,16,32) with $SQNR_{min}$ set to 15dB*

training data symbols, through the channels `sym_tr_re` and `sym_tr_im` for the real and imaginary components respectively, and writes out 48 channel coefficients to the channels `chan_coeff_re` and `chan_coeff_im` for the real and imaginary components respectively. The second module in the design is the Alamouti decoder, which reads in 1060 data symbols, through the channels `sym_data_re` and `sym_data_im` for the real and imaginary components respectively, and writes out 960 output symbols to the channels `sym_out_re` and `sym_out_im` for the real and imaginary components respectively. Additionally to the SystemC model of the receiver itself, a set of floating-point test patterns, corresponding to five functional simulations of the system, is also supplied to the *fixify* environment.

Automated conversion is performed by the *fixify* environment on the MIMO receiver design, using all the three available optimization methods. Firstly, restricted-set full search

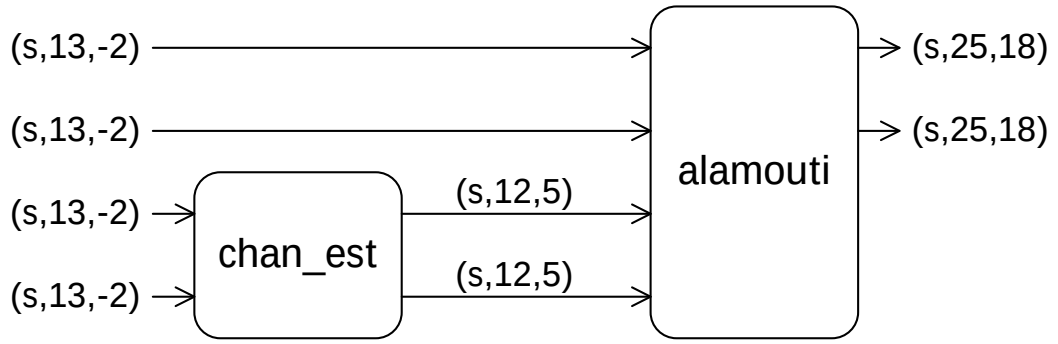

Figure 4.4: *An example optimization result: full search(8,16,32) with SQNR$_{min}$ set to 15dB*

with the possible bitwidth set of $\{8, 16, 32\}$ is used, corresponding to the available set of fixed-point formats on the TIC6416 DSP which is used in the original implementation [53]. The second and third optimization methods were greedy and branch-and-bound, respectively. Each optimization method is employed with 20 different error limits, corresponding to the SQNR$_{min}$ range of [5dB,100dB], in 5dB steps. For each of the error limits, each of the optimization algorithms returns a fixed-point configuration as a result. An example optimization result (full search optimization, set of possible bitwidths $\{8, 16, 32\}$, SQNR$_{min}$ set to 15dB) is given in Table 4.3, and depicted in Figure 4.4. Please refer to Appendix E for a full and detailed listing of all the optimization results. From these results, conclusions can be drawn about the relative performance of each optimization method, both in terms of the quality of its results and its runtime performance.

The trade-off between the implementation cost of the system, c, and the degradation in numeric performance through the introduction of fixed-point formats, represented in terms of SQNR$_{min}$, is shown in Figure 4.5 for each of the three optimization methods. Also represented in Figure 4.5 is the fixed-point configuration determined by the designer manually, for comparison with the automated optimization results. Please note that the example optimization result shown in Table 4.3 and in Figure 4.4 is represented

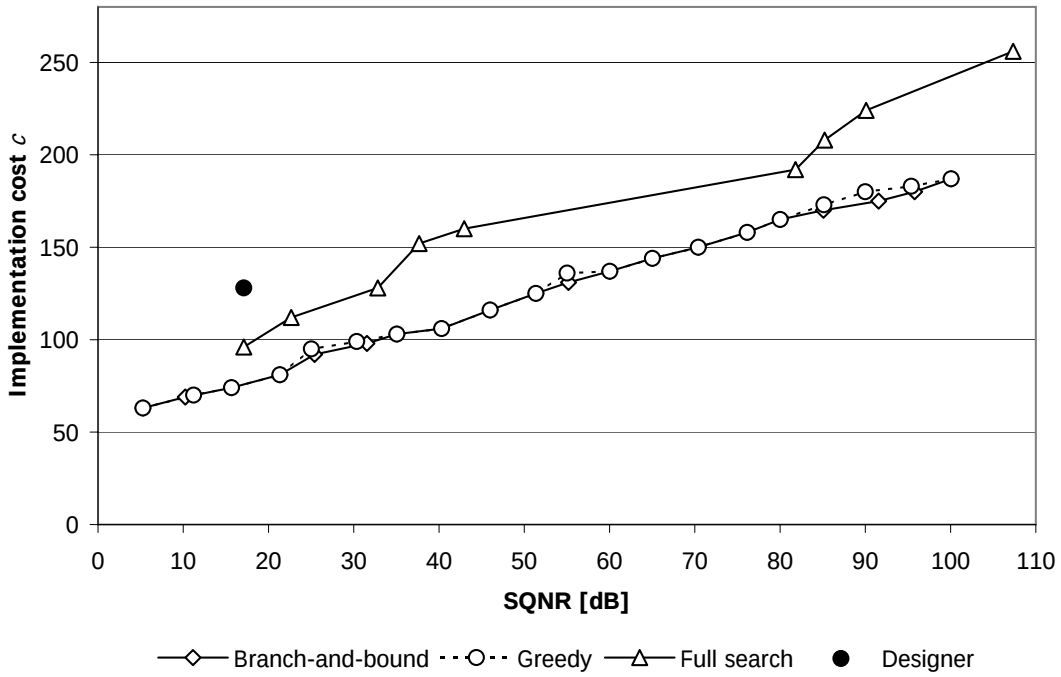

Figure 4.5: *Optimization results for the MIMO receiver design*

in Figure 4.5 as a single point, returned by the full search algorithm and located at (17,09dB,96).

It can immediately be noted from Figure 4.5 that all three optimization methods generally require increased implementation cost with increasing SQNR requirements, as is intuitive. In other words, the optimization algorithms are able to find fixed-point configurations with lower implementation costs when more degradation of numeric performance is allowed.

It can also be noted from Figure 4.5 that the optimization results of the restricted-set full search algorithm consistently (i.e. over the entire examined range [5dB,100dB]) require higher implementation costs for the same level of numeric performance then both the greedy and the branch-and-bound optimization algorithms. The reason for this effect is the restricted set of possible bitwidths that the full search algorithm can assign to each data channel. For this reason, the full search algorithm can only move through the

solution space in large quantum steps, thus not being able to fine tune the fixed-point format of each channel. On the other hand, greedy and branch-and-bound algorithms both have full freedom to assign any positive integer (strictly greater than zero) as the word length of the fixed-point format for each channel in the design, thus consistently being able to extract fixed-point configurations with lower implementation costs for the same SQNR levels.

Also, Figure 4.5 shows that, though the branch-and-bound algorithm consistently finds the fixed-point configuration with the lowest implementation cost for a given level of SQNR, the greedy algorithm performs only slightly worse. In 13 out of the 20 optimizations, the greedy algorithm returned the same fixed-point configuration as the branch-and-bound algorithm. In the other seven cases, the sub-tree relaxation routine of the branch-and-bound algorithm discovered a superior fixed-point configuration. In these cases, the relative improvement (reduced implementation cost) by using the branch-and-bound algorithm ranged between 1,02% and 3,82%.

Furthermore, it can be noted that the fixed-point configuration found by the designer manually can be improved for both the DSP implementation (i.e. with the restricted-set full search algorithms) and the custom hardware implementation (i.e. with the greedy and/or branch-and-bound algorithms). The designer optimized the design to the fixed-point configuration where all the word lengths are set to 16 bits by manual trial and error, as is traditionally the case. After confirming that the design has satisfactory performance with all word lengths set to 32 bits, the designer assigned all the word lengths to 16 bits and found that this configuration also performs satisfactorily. However, it is possible to obtain lower implementation cost for the same SQNR level, as well as superior numeric performance (i.e. higher SQNR) for the same implementation cost, as can be seen in Figure 4.5.

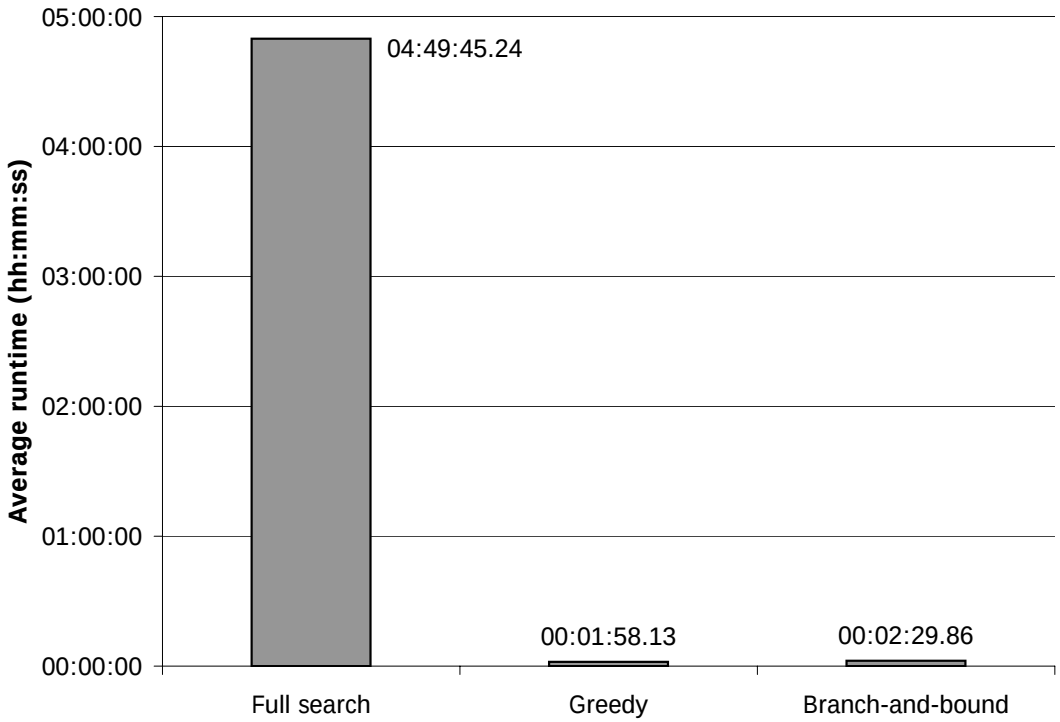

Figure 4.6: *Runtime performance of the fixify environment on the MIMO receiver design*

Another key aspect of the optimization algorithms in the *fixify* environment is their runtime performance. The comparison of average runtimes of the three optimization algorithms is shown graphically in Figure 4.6. It can immediately be noted that the full search algorithm requires significantly longer runtimes than both the greedy and the branch-and-bound algorithms. This is of course due to the indiscriminate search though the solution space carried out by the full search algorithm, whereas the greedy and branch-and-bound algorithms intelligently traverse the solution space, thus minimizing the number of required simulations.

It can also be noted from Figure 4.6 that the branch-and-bound algorithm requires on average 28,86% more time to optimize the fixed-point formats in the system, due to the sub-branch relaxation routine. In other words, the 1,02%-3,82% improvement in performance mentioned earlier directly corresponds to a 28,86% increase in the required runtime.

4.5 Conclusions

The *fixify* environment has been successfully applied to performing a fully automated floating-point to fixed-point conversion, thereby completely replacing the designer's manual effort. Furthermore, it has been applied to an industrial DSP design, with runtimes in the order of minutes, thus proving practicality of its application.

For designs that are to be mapped to software running on a DSP core, restricted-set full search is the best choice of optimization technique, since it offers guaranteed optimal results and optimizes the design directly to the set of fixed-point bitwidths that are native to the DSP core in question. For custom hardware implementations, the best choice of optimization option is the branch-and-bound algorithm, offering guaranteed optimal results for system with no feedback cycles and monotonic performance functions. However, for high-complexity designs with relatively long simulation times, the greedy search algorithm is an excellent alternative, offering considerably reduced optimization runtimes, with very little sacrifice in the quality of results.

5. CONCLUSIONS

This thesis presents OTIE, an open environment for flexible integration of EDA tools, aimed at combating the inefficiencies of the modern design process for embedded systems. The OTIE is based on a single description of the system to which all EDA tools used by all the design teams interface. This single description of the system, the SSD, is implemented based on a MySQL database, offering a safe, performant, and flexible implementation for this repository of refinement information.

The unique contribution of the presented environment is its ability to transcend the interoperability problems inherent in modern EDA tools, thus resolving the fundamental bottleneck present in the modern design processes for embedded systems. This ability of the OTIE has been shown through the integration and use of the VP and the *fixify* environments, successfully automating multiple parts of the design process using the same description of the system resident in the SSD. Thus the OTIE overcomes the customary inefficiencies which arise in traditional design flows, including constant rewriting of system descriptions, inefficient feedback of debugging information, the extremely fragmented EDA tool support, and the rapid emergence of new EDA tools.

Another unique contribution presented in this thesis is the consistent and flexible implementation of the OTIE, allowing seamless incorporation of commercial, academic, and in-house developed EDA tools. As such, this implementation is independent of any particular EDA vendor, development environment, or design language.

One original feature of the OTIE is its ability to unite all the system descriptions used during the development of the system into one central repository, the SSD. Hence, for the first time, all the teams involved in the design process can share all the refinement information for the system under development, with transparent and simultaneous access.

This in turn eliminates the need for the traditional rewriting of system descriptions, which has several positive effects. Firstly, by eliminating the need to invest design effort into the rewriting of system descriptions, the OTIE raises the efficiency of the design process. Also, the quality of the design process is increased, since the coding errors introduced by the designer during manual recoding have also been eliminated. A further improvement in the quality of the design process is gained by the OTIE through improved feedback of design errors. Since all the teams share the description of the system in the SSD, reporting bugs backwards in the development process is simplified.

Finally, a key feature of the OTIE is its unique ability to incorporate new EDA tools as they become available. Since many parts of the design process are still performed manually, completely new EDA tools are constantly emerging in the research as well as commercial arenas. Through the integration of the *fixify* environment - a novel EDA tool designed to tackle a previously unautomated task - the ability of the OTIE to incorporate successfully new and previously unavailable EDA tools has been demonstrated. This feature also makes the OTIE "future proof" and able to adapt to the changing landscape of EDA technology.

A number of future developments of the OTIE can also be identified. These include the ongoing addition of commercial and research tools which are integrated into the OTIE. The OTIE is capable of supporting, and in the future may include EDA tools for design of analogue and mixed-signal circuits, high-level synthesis of reconfigurable cir-

cuits, architecture mapping, and others. Also, the integrated VP environment currently supports Cossap algorithmic models, as a proof of concept, but future developments of this environment may include support for models described in any other design language (e.g. SystemC). Future work on the *fixify* environment may similarly include support for floating-point models described in languages other than SystemC, as well as provision of additional optimization algorithms. Also, future developments of this environment may enable optimization of local process variables as well, in addition to the inter-process data channels which are currently the target of optimization. Furthermore, following an evaluation of the sensitivity of the optimization results to the test-coverage afforded by the supplied floating-point test vectors, it may be possible to offer automated generation of this data, thus further disburdening the designer and making the conversion process even more robust. By adding the ability to set individual E_{lim} values for each data channel in the system (rather than a uniform E_{lim} value for the whole design), even more freedom in exploration can be given to the designer. Finally, further refinement of cost functions, for example by relative weighting of data channels in the system, or even individual bits therein, will lead to increasingly realistic estimates of the final implementation cost and thus improved quality of optimization results.

BIBLIOGRAPHY

[1] M. Broy, "Automotive Software Engineering," in *IEEE International Conference on Software Engineering*, pp. 719–720, May 2003.

[2] Y. Neuvo, "Cellular Phones as Embedded Systems," in *IEEE International Solid-State Circuits Conference ISSCC'04*, pp. 32–37, February 2004.

[3] J. Hausner and R. Denk, "Implementation of Signal Processing Algorithms for 3G and Beyond," *IEEE Microwave And Wireless Components Letters*, vol. 13, no. 8, 2003.

[4] SPIRIT Consortium. `www.spiritconsortium.com`.

[5] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," in *Proceedings of the IEEE*, vol. 91, pp. 145–164, January 2003.

[6] P. Belanović, M. Holzer, D. Mičušík, and M. Rupp, "Design Methodology of Signal Processing Algorithms in Wireless Systems," in *International Conference on Computer, Communication and Control Technologies CCCT'03*, (Orlando, FL, USA), pp. 288–291, July 2003.

[7] P. Belanović, M. Holzer, B. Knerr, M. Rupp, and G. Sauzon, "Automatic Generation of Virtual Prototypes," in *International Workshop on Rapid System Prototyping RSP'04*, (Geneva, Switzerland), pp. 114–118, June 2004.

[8] P. Belanović and M. Holzer and B. Knerr and M. Rupp, "Automated Verification Pattern Refinement for Virtual Prototypes," in *Conference of Design of Circuits and Integrated Systems DCIS'05*, (Lisbon, Portugal), November 2005.

[9] P. Belanović and M. Rupp, "Fixify: A Toolset for Automated Floating-point to Fixed-point Conversion," in *International Conference on Computer, Communication and Control Technologies CCCT'04*, vol. VIII, (Austin, TX, USA), pp. 28–29, August 2004.

[10] P. Belanovic and M. Rupp, "Automated Floating-point to Fixed-point Conversion with the *fixify* Environment," in *International Workshop on Rapid System Prototyping RSP'05*, (Montreal, Canada), pp. 172–178, June 2005.

[11] F. Langerak and E. Hultink, "The Impact of New Product Development Acceleration Approaches on Speed and Profitability: Lessons for Pioneers and Fast Followers," *IEEE Transactions on Engineering Management*, vol. 52, no. 1, pp. 30–42, 2005.

[12] International Engineering Consortium, "Universal Mobile Telecommunications System (UMTS) Protocols and Protocol Testing," August 2004. `www.iec.org/online/tutorials/umts`.

[13] GSM Association, "Quarterly statistics q1 2005." `www.gsm.org/news/statistics/pdf/gsma_stats_q1_05.pdf`.

[14] R. Subramanian, "Shannon vs. Moore: Driving the Evolution of Signal Processing Platforms in Wireless Communications," in *IEEE Workshop on Signal Processing Systems SIPS'02*, October 2002.

[15] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, pp. 114–117, April 1965.

[16] International SEMATECH, "International Technology Roadmap for Semiconductors," 1999. `www.sematech.org`.

[17] M. Rupp, A. Burg, and E. Beck, "Rapid Prototyping for Wireless Designs: the Five-Ones Approach," *Signal Processing 2003*, June 2003.

[18] R. S. Janka, *Specification and Design Methodology for Real-Time Embedded Systems*. Kluwer Academic Publishers, 2002. ISBN 0-7923-7626-9.

[19] SystemVerilog. `www.systemverilog.org`.

[20] Open SystemC Initiative. `www.systemc.org`.

[21] SPIRIT Schema Working Group Membership, "SPIRIT-User Guide v1.1," tech. rep., SPIRIT Consortium, June 2005.

[22] G. Karsai, "Design Tool Integration: An Exercise in Semantic Interoperability," in *Proceedings of the IEEE Engineering of Computer Based Systems*, (Edinburg, UK), March 2000.

[23] Synopsys Inc., "Galaxy Design Platform." `www.synopsys.com/products/solutions/galaxy_platform.html`.

[24] MySQL Database Products. `www.mysql.com/products/database`.

[25] Concurrent Versions System (CVS). `www.nongnu.org/cvs`.

[26] IBM Rational ClearCase. `www-306.ibm.com/software/awdtools/clearcase`.

[27] M. Holzer and M. Rupp, "Static Code Analysis of Functional Descriptions in SystemC," in *IEEE International Workshop on Electronic Design, Test and Applications*, (Kuala Lumpur), January 2006.

[28] T. Anderson and R. Schutten and F. Thoen, "Virtual Prototypes Cut Software Bottleneck," tech. rep., Wireless Systems Design Online Magazine, February 2005. `www.wsdmag.com/Articles/ArticleID/9821`.

[29] C. Hein, J. Pridgen, and W. Kleine, "RASSP Virtual Prototyping of DSP Systems," in *Design Automation Conference DAC'97*, pp. 492–497, 1997.

[30] J. Cockx, "Efficient Modelling of Preemption in Virtual Prototype," in *International Workshop on Rapid System Prototyping RSP'00*, (Paris, France), pp. 14–19, June 2000.

[31] A. Hoffmann, T. Kogel, and H. Meyr, "A Framework for Fast Hardware-Software Co-simulation," in *Design, Automation and Test in Europe DATE'01*, (Munich, Germany), pp. 760–765, 2001.

[32] A. Hemani, A. K. Deb, J. Öberg, A. Postula, D. Lindqvist, and B. Fjellborg, "System Level Virtual Prototyping of DSP SOCs Using Grammar Based Approach," *Design Automation for Embedded Systems*, vol. 5, no. 3, pp. 295–311, 2000.

[33] U. Bortfeld and C. Mielenz, "Whitepaper C++ System Simulation Interfaces," July 2000.

[34] P. Varma and S. Bhatia, "A Structured Re-Use Methodology for Core-Based System Chips," in *IEEE International Test Conference ITC'98*, pp. 294–302, 1998.

[35] B. Stöhr, M. Simmons, and J. Geishauser, "FlexBench: Reuse of Verification IP to Increase Productivity," in *Design, Automation and Test In Europe DATE'02*, p. 1131, 2002.

[36] Odin Technology, "Axe Automated Testing Framework," 2004. `www.odin.co.uk/downloads/AxeFlyer.pdf`.

[37] P. Belanović, B. Knerr, M. Holzer, G. Sauzon, and M. Rupp, "A consistent design methodology for wireless embedded systems," *EURASIP Journal of Applied Signal Processing, Special Issue on DSP Enabled Radio*, pp. 2598–2612, September 2005.

[38] B. Knerr, M. Holzer, P. Belanović, G. Sauzon, and M. Rupp, "Advanced UMTS Receiver Chip Design Using Virtual Prototyping," in *International Symposium on Signals, Systems, and Electronics ISSSE'04*, (Linz, Austria), August 2004.

[39] P. Belanović, B. Knerr, M. Holzer, and M. Rupp, "A Fully Automated Environment for Verification of Virtual Prototypes," *EURASIP Journal on Applied Signal Processing Special Issue on Design Methods for DSP Systems*, 2006. (to appear).

[40] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogenous Systems," *International Journal in Computer Simulation*, vol. 4, no. 2, 1994.

[41] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A Fixed-Point Design and Simulation Environment," in *Design, Automation and Test In Europe DATE'98*, pp. 429–435, February 1998.

[42] S. Kim, K. Kum, and W. Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 45, pp. 1455–1464, November 1998.

[43] I. Synopsys, "Converting ANSI-C into Fixed-Point Using CoCentric Fixed-Point Designer," April 2000.

[44] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," in *SIGPLAN Conference on Program Language Design and Implementation PLDI'00*, pp. 108–120, June 2000.

[45] D. Menard, D. Chillet, F. Charot, and O. Sentieys, "Automatic Floating-point to Fixed-point Conversion for DSP Code Generation," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems CASES'02*, (Grenoble, France), pp. 270–276, October 2002.

[46] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, Accurate Static Analysis for Fixed-point Finite Precision Effects in DSP Designs," in *International Conference on Computer Aided Design*, (San Jose, CA, USA), November 2003.

[47] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens, "A Methodology and Design Environment for DSP ASIC Fixed Point Refinement," in *Design, Automation and Test in Europe Conference DATE'99*, (Munich, Germany), pp. 271–276, March 1999.

[48] Y. Cao and H. Yasuura, "Quality-Driven Design by Bitwidth Optimization for Video Applications," in *IEEE/ACM Asia and South Pacific Design Automation Conference*, pp. 532–537, January 2003.

[49] C. Shi and R. W. Brodersen, "An Automated Floating-point to Fixed-point Conversion Methodology," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 529–532, April 2003.

[50] MathWorks Simulink. `www.mathworks.com/products/simulink`.

[51] Open SystemC Initiative, "Draft Standard SystemC Language Reference Manual," April 2005.

[52] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, 2000. ISBN 0-07-013143-0.

[53] C. Mehlführer, F. Kaltenberger, M. Rupp, and G. Humer, "A Scalable Rapid Prototyping System for Real-time MIMO OFDM Transmission," in *Conference on DSP Enabled Radio*, (Southampton, UK), September 2005.

[54] M. Holzer, P. Belanović, and M. Rupp, "A Consistent Design Methodology to Meet SDR Challenges," in *Wireless World Research Forum WWRF9*, (Zurich, Switzerland), July 2003.

[55] M. Holzer, B. Knerr, P. Belanović, and M. Rupp, "Faster Complex SoC Design by Virtual Prototyping," in *International Conference on Cybernetics and Information Technologies, Systems, and Applications CITSA'04*, (Orlando, FL, USA), pp. 305–309, July 2004.

[56] B. Knerr, M. Holzer, and M. Rupp, "HW/SW Partitioning Using High Level Metrics," in *International Conference on Computer, Communication and Control Technologies CCCT'04*, vol. VIII, (Austin, TX, USA), pp. 33–38, August 2004.

[57] M. Coors, H. Keding, O. Lüthje, and H. Meyr, "Design and DSP Implementation of Fixed-point Systems," in *EURASIP Journal of Applied Signal Processing*, no. 9, pp. 908–925, September 2002.

[58] M. Stephenson, "Bitwise: Optimizing Bitwidths Using Data-Range Propagation," Master's thesis, Massachusetts Institute of Technology, May 2000.

[59] G. Caffarena, A. Fernandez, C. Carreras, and O. Nieto-Taladriz, "Fixed-point Refinement of OFDM-based Adaptive Equalizers: A Heuristic Approach," in *European Signal Processing Conference EUSIPCO 2004*, (Vienna, Austria), September 2004.

[60] W3 Extensible Markup Language (XML). `www.w3.org/XML`.

[61] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002. ISBN 1-4020-7072-1.

[62] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl*. O'Reilly, 3 ed., 2000. ISBN 0-596-00027-8.

[63] L. Lamport, *LaTeX: a Document Preparation System*. Addison Wesley Longman, 2 ed., December 1996. ISBN 0-201-52983-1.

APPENDICES

A. LIST OF ACRONYMS

3G	Third Generation
ANSI	American National Standards Institute
CAGR	Compounded Annual Growth rate
CCF	Corner Case Finder
CDMA	Code Division Multiple Access
CFG	Control Flow Graph
CGI	Common Gateway Interface
CVS	Concurrent Versioning System
DAG	Directed Acyclic Graph
DDF	Dynamic Data Flow
DFG	Data Flow Graph
DMA	Direct Memory Access
DSP	Digital Signal Processor
DUD	Decoding of User Data
DUV	Device Under Verification
EDA	Electronic Design Automation
EDGE	Enhanced Data rate for GSM Evolution
FCO	Fixed-point Configuration Optimizer
FIFO	First-In, First-Out
GLOCCT	GLObal Control, Configuration and Timing
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
HA	Hardware Accelerator
HDL	Hardware Description Language
HSCSD	High Speed Circuit Switched Data
HSG	Hybrid System Generator
HTML	Hyper Text Markup Language
IF	Intermediate Format
IMS	Integrated Model Server
I/O	Input/Output
IP	Intellectual Property
MIC	Model Integrated Computing
MIMO	Multiple-Input, Multiple-Output
OTIE	Open Tool Integration Environment
RAM	Random Access Memory
RTL	Register Transistor Level
SCE	System Cost Estimator

SDF	Synchronous Data Flow
SDI	System Description Interface
SDL	Specification Description Language
SoC	System on Chip
SPE	System Performance Estimator
SPIRIT	Structure for Packaging, Integrating and Re-Using IP within Tool-flows
SQL	Strctured Query Language
SQNR	Signal to Quantization Noise Ratio
SSD	Single System Description
TGS	Test Generator Script
TDMA	Time Division Multiple Access
TLM	Transaction Level Modelling
UML	Universal Modelling Language
UMTS	Universal Mobile Terrestrial System
UWB	Ultra Wide Band
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Cicruit
VP	Virtual Prototype
VPG	Virtual Prototype Generator
VSIA	Virtual Socket Interface Alliance
WLAN	Wireless Local Area Network
XML	eXtensible Markup Language

B. CAGR CALCULATIONS

The Compounded Annual Growth Rate (CAGR) is a measure of the average growth in the value of a variable per year, over a number of years, taking into consideration the effects of compounding. It is expressed as follows.

$$CAGR = \left(\frac{End\_value}{Start\_value}\right)^{\frac{1}{years}} - 1$$

Since the algorithmic complexity of wireless communications systems experienced a growth factor of 10^6 over the last 25 years, this can be expressed in terms of the CAGR as follows.

$$\begin{aligned} CAGR &= \left(10^6\right)^{\frac{1}{25}} - 1 \\ &= 73,8\% \end{aligned}$$

Similarly, Moore's law dictates the growth in complexity of silicon integrated circuits, and predicts doubling of complexity every 18 months. Hence, the CAGR of Moore's law can be calculated as follows.

$$\begin{aligned} CAGR &= (2)^{\frac{1}{1,5}} - 1 \\ &= 58,7\% \end{aligned}$$

C. XML INTERMEDIATE FORMAT

The IF representation of the system is based on XML, as described in Section 2.5.2. The XML tags used to represent the system are listed below, together with a short description of each. Please also refer to Figure C.1 for a graphical representation of the XML structure. Following this, the IF representation of the `CellSearcher` design is given, to further illustrate the XML format of the IF representation.

`project` Contains an entire project, consisting of one or more designs.

`design` Contains one design, made up of instances of modules.

`modules` A section inside the `design` tag, where module definitions are given.

`module` Contains the definition of a module.

`submod` Defines sub-modules of structural modules.

`signal` Defines data channels connecting sub-modules.

`connections` A section inside the `module` tag or the `block` tag. When inside the `module` tag, contains definitions of sub-module connections. When inside the `block` tag, contains definitions of predecessor/successor connections of a basic block.

`connection` When inside the `module` tag, describes a connection between sub-modules through a data channel. When inside the `block` tag, describes a connection to another basic block, either as a predecessor or successor.

`port` Defines a port through which a functional module receives and/or sends data.

`type` Defines the type of a port (in, out, in/out,...).

`datatype` Defines the datatype of a port.

`process` Contains the definition of a process within a functional module.

`blocks` A section inside the `process` tag where basic blocks are defined.

`block` Contains the definition of a basic block.

`properties` A section inside the `block` tag or the `process` tag. When inside the `block` tag, contains definitions of properties relevant to basic blocks. When inside the `process` tag, contains definitions of properties relevant to processes.

`property` When inside the `block` tag, describes a a property relevant to basic blocks. When inside the `process` tag, describes a property relevant to processes.

`instances` A section inside the `design` tag, where module instantiations are given.

`instance` Defines an instance of a module.

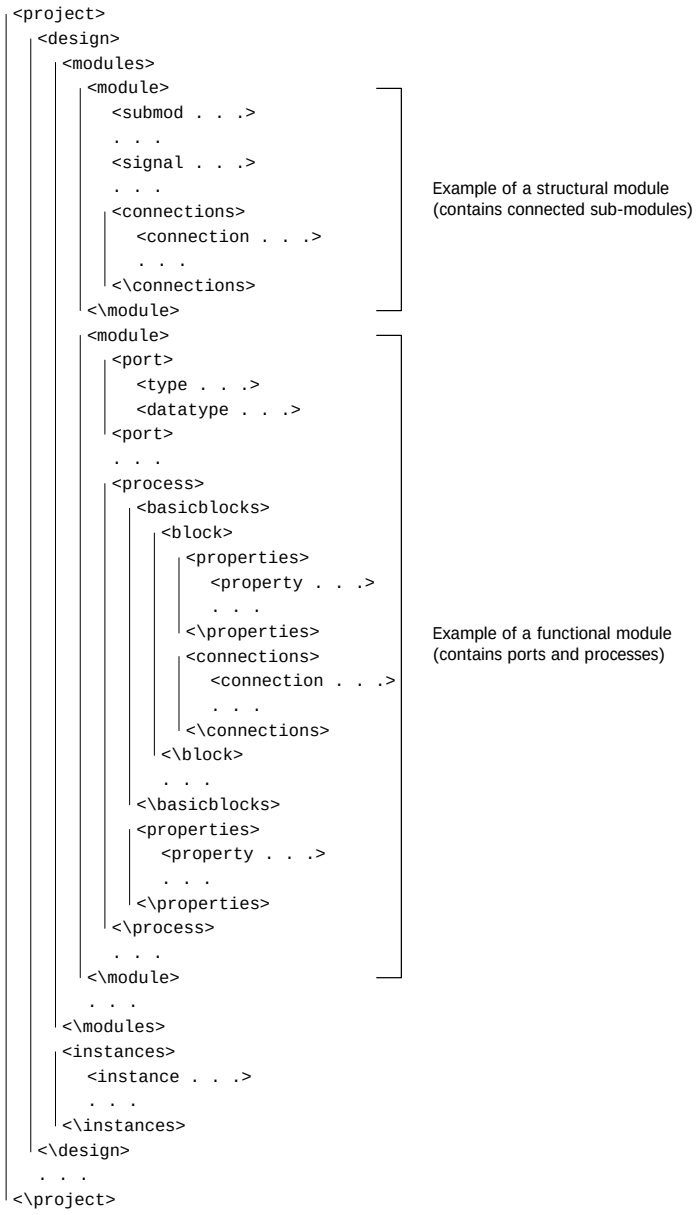

Figure C.1: *Structure of the IF representation of the system*

The following is the IF representation of the `CellSearcher` design, illustrating the XML format of the IF representation. Please note that in the interest of brevity, this

IF representation is shortened, that is to say some parts of the IF representation are omitted. The information on basic blocks and their contents is shown only for one of the functional modules in the design, the `MatchedFilter` module. While analogous information for all the other functional modules in the design is available as well, it is omitted here.

```
<project name="Cellsearcher" id="cdl_prj_id_000000001">
  <design>
    <modules>
      <module name="Cellsearcher" id="cdl_mod_id_000000001">
        <submod module_id="cdl_mod_id_000000002" name="SqrAndSum" />
        <submod module_id="cdl_mod_id_000000003" name="MatchedFilter" />
        <submod module_id="cdl_mod_id_000000004" name="Display" />
        <submod module_id="cdl_mod_id_000000005" name="PeakDetector" />
        <submod module_id="cdl_mod_id_000000006" name="SlotAccu" />
        <submod module_id="cdl_mod_id_000000007" name="FrameSource" />
        <signal name="I" />
        <signal name="Q" />
        <signal name="filtered_I" />
        <signal name="filtered_Q" />
        <signal name="Energy" />
        <signal name="Acc_Energy" />
        <signal name="PeakIndex" />
        <signal name="PeakHeight" />
        <connections>
          <connection sub_mod_name="FrameSource"
                      sub_port="out_I"
                      signal="I" />
          <connection sub_mod_name="MatchedFilter"
                      sub_port="in_I"
                      signal="I" />
          <connection sub_mod_name="FrameSource"
                      sub_port="out_Q"
                      signal="Q" />
          <connection sub_mod_name="MatchedFilter"
                      sub_port="in_Q"
                      signal="Q" />
          <connection sub_mod_name="MatchedFilter"
                      sub_port="out_I"
                      signal="filtered_I" />
          <connection sub_mod_name="SqrAndSum"
                      sub_port="in_I"
                      signal="filtered_I" />
          <connection sub_mod_name="MatchedFilter"
                      sub_port="out_Q"
                      signal="filtered_Q" />
          <connection sub_mod_name="SqrAndSum"
                      sub_port="in_Q"
                      signal="filtered_Q" />
          <connection sub_mod_name="SqrAndSum"
                      sub_port="out_En"
                      signal="Energy" />
          <connection sub_mod_name="SlotAccu"
                      sub_port="in_En"
                      signal="Energy" />
          <connection sub_mod_name="SlotAccu"
                      sub_port="out_Accu_En"
                      signal="Acc_Energy" />
          <connection sub_mod_name="PeakDetector"
                      sub_port="in_En"
                      signal="Acc_Energy" />
          <connection sub_mod_name="PeakDetector"
```

```
                sub_port="out_Peak_Index"
                signal="PeakIndex" />
    <connection sub_mod_name="Display"
                sub_port="in_PI"
                signal="PeakIndex" />
    <connection sub_mod_name="PeakDetector"
                sub_port="out_Peak_Height"
                signal="PeakHeight" />
    <connection sub_mod_name="Display"
                sub_port="in_PH"
                signal="PeakHeight" />
  </connections>
</module>
<module name="MatchedFilter" id="cdl_mod_id_000000003">
  <port name="in_I">
    <type>in</type>
    <datatype>double</datatype>
  </port>
  <port name="in_Q">
    <type>in</type>
    <datatype>double</datatype>
  </port>
  <port name="out_I">
    <type>out</type>
    <datatype>double</datatype>
  </port>
  <port name="out_Q">
    <type>out</type>
    <datatype>double</datatype>
  </port>
  <process id="cdl_pro_id_000000001" name="filterFunc">
    <blocks>
      <block name="bb0">
        <properties>
          <property name="DfgDepth">0</property>
          <property name="NrOfAdd">0</property>
          <property name="NrOfSub">0</property>
          <property name="NrOfMul">0</property>
          <property name="NrOfIf">0</property>
          <property name="NrOfXor">0</property>
          <property name="NrOfSL">0</property>
          <property name="NrOfSR">0</property>
          <property name="NrOfAnd">0</property>
          <property name="NrOfOr">0</property>
          <property name="NrOfDiv">0</property>
        </properties>
        <connection type="succ">bb1</connection>
      </block>
      <block name="bb1">
        <properties>
          <property name="DfgDepth">0</property>
          <property name="NrOfAdd">0</property>
          <property name="NrOfSub">0</property>
          <property name="NrOfMul">0</property>
          <property name="NrOfIf">0</property>
          <property name="NrOfXor">0</property>
          <property name="NrOfSL">0</property>
          <property name="NrOfSR">0</property>
          <property name="NrOfAnd">0</property>
          <property name="NrOfOr">0</property>
          <property name="NrOfDiv">0</property>
        </properties>
        <connection type="succ">bb2</connection>
        <connection type="pred">bb0</connection>
```

```
</block>
<block name="bb2">
  <properties>
    <property name="Loopcountupper">16</property>
    <property name="Loopcountlower">16</property>
    <property name="Loopcountstep">1</property>
    <property name="Loopcountbitsize">32</property>
    <property name="DfgDepth">0</property>
    <property name="NrOfAdd">0</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb3</connection>
  <connection type="pred">bb1</connection>
  <connection type="pred">bb10</connection>
</block>
<block name="bb3">
  <properties>
    <property name="Loopcountupper">16</property>
    <property name="Loopcountlower">16</property>
    <property name="Loopcountstep">1</property>
    <property name="Loopcountbitsize">32</property>
    <property name="DfgDepth">2</property>
    <property name="NrOfAdd">3</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">4</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb5</connection>
  <connection type="succ">bb4</connection>
  <connection type="pred">bb2</connection>
  <connection type="pred">bb6</connection>
</block>
<block name="bb4">
  <properties>
    <property name="DfgDepth">3</property>
    <property name="NrOfAdd">4</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">2</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb6</connection>
  <connection type="pred">bb3</connection>
</block>
<block name="bb5">
```

```
  <properties>
    <property name="DfgDepth">3</property>
    <property name="NrOfAdd">2</property>
    <property name="NrOfSub">2</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">2</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb6</connection>
  <connection type="pred">bb3</connection>
</block>
<block name="bb6">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb3</connection>
  <connection type="succ">bb7</connection>
  <connection type="pred">bb4</connection>
  <connection type="pred">bb5</connection>
</block>
<block name="bb7">
  <properties>
    <property name="DfgDepth">0</property>
    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">1</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb9</connection>
  <connection type="succ">bb8</connection>
  <connection type="pred">bb6</connection>
</block>
<block name="bb8">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">2</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
```

```
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb10</connection>
  <connection type="pred">bb7</connection>
</block>
<block name="bb9">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">0</property>
    <property name="NrOfSub">2</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">0</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb10</connection>
  <connection type="pred">bb7</connection>
</block>
<block name="bb10">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">0</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb2</connection>
  <connection type="succ">bb11</connection>
  <connection type="pred">bb8</connection>
  <connection type="pred">bb9</connection>
</block>
<block name="bb11">
  <properties>
    <property name="DfgDepth">1</property>
    <property name="NrOfAdd">1</property>
    <property name="NrOfSub">0</property>
    <property name="NrOfMul">0</property>
    <property name="NrOfIf">1</property>
    <property name="NrOfXor">0</property>
    <property name="NrOfSL">1</property>
    <property name="NrOfSR">0</property>
    <property name="NrOfAnd">0</property>
    <property name="NrOfOr">0</property>
    <property name="NrOfDiv">0</property>
  </properties>
  <connection type="succ">bb13</connection>
  <connection type="succ">bb12</connection>
  <connection type="pred">bb10</connection>
</block>
<block name="bb12">
  <properties>
    <property name="DfgDepth">0</property>
    <property name="NrOfAdd">0</property>
```

```
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">0</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb13</connection>
    <connection type="pred">bb11</connection>
  </block>
  <block name="bb13">
    <properties>
      <property name="DfgDepth">1</property>
      <property name="NrOfAdd">1</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">1</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">1</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb15</connection>
    <connection type="succ">bb14</connection>
    <connection type="pred">bb11</connection>
    <connection type="pred">bb12</connection>
  </block>
  <block name="bb14">
    <properties>
      <property name="DfgDepth">0</property>
      <property name="NrOfAdd">0</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">0</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb15</connection>
    <connection type="pred">bb13</connection>
  </block>
  <block name="bb15">
    <properties>
      <property name="DfgDepth">0</property>
      <property name="NrOfAdd">0</property>
      <property name="NrOfSub">0</property>
      <property name="NrOfMul">0</property>
      <property name="NrOfIf">0</property>
      <property name="NrOfXor">0</property>
      <property name="NrOfSL">0</property>
      <property name="NrOfSR">0</property>
      <property name="NrOfAnd">0</property>
      <property name="NrOfOr">0</property>
      <property name="NrOfDiv">0</property>
    </properties>
    <connection type="succ">bb16</connection>
```

```
          <connection type="pred">bb13</connection>
          <connection type="pred">bb14</connection>
        </block>
        <block name="bb16">
          <properties>
            <property name="DfgDepth">0</property>
            <property name="NrOfAdd">0</property>
            <property name="NrOfSub">0</property>
            <property name="NrOfMul">0</property>
            <property name="NrOfIf">0</property>
            <property name="NrOfXor">0</property>
            <property name="NrOfSL">0</property>
            <property name="NrOfSR">0</property>
            <property name="NrOfAnd">0</property>
            <property name="NrOfOr">0</property>
            <property name="NrOfDiv">0</property>
          </properties>
          <connection type="pred">bb15</connection>
        </block>
      </blocks>
      <properties>
        <property name="NrOfAdd">16</property>
        <property name="NrOfSub">4</property>
        <property name="NrOfMul">0</property>
        <property name="NrOfIf">6</property>
        <property name="NrOfXor">0</property>
        <property name="NrOfSL">11</property>
        <property name="NrOfSR">0</property>
        <property name="NrOfAnd">0</property>
        <property name="NrOfOr">0</property>
        <property name="NrOfDiv">0</property>
      </properties>
    </process>
  </module>
  <module name="SqrAndSum" id="cdl_mod_id_000000002">
    <port name="in_I">
      <type>in</type>
      <datatype>double</datatype>
    </port>
    <port name="in_Q">
      <type>in</type>
      <datatype>double</datatype>
    </port>
    <port name="out_En">
      <type>out</type>
      <datatype>double</datatype>
    </port>
  </module>
  <module name="Display" id="cdl_mod_id_000000004">
    <port name="in_PH">
      <type>in</type>
      <datatype>double</datatype>
    </port>
    <port name="in_PI">
      <type>in</type>
      <datatype>int</datatype>
    </port>
  </module>
  <module name="PeakDetector" id="cdl_mod_id_000000005">
    <port name="in_En">
      <type>in</type>
      <datatype>double</datatype>
    </port>
    <port name="out_Peak_Height">
```

```
        <type>out</type>
        <datatype>double</datatype>
      </port>
      <port name="out_Peak_Index">
        <type>out</type>
        <datatype>int</datatype>
      </port>
    </module>
    <module name="SlotAccu" id="cdl_mod_id_000000006">
      <port name="in_En">
        <type>in</type>
        <datatype>double</datatype>
      </port>
      <port name="out_Accu_En">
        <type>out</type>
        <datatype>double</datatype>
      </port>
    </module>
    <module name="FrameSource" id="cdl_mod_id_000000007">
      <port name="out_I">
        <type>out</type>
        <datatype>double</datatype>
      </port>
      <port name="out_Q">
        <type>out</type>
        <datatype>double</datatype>
      </port>
    </module>
  </modules>
  <instances>
    <instance id="cdl_ins_id_000000001"
              name="CellSearcher_1"
              module_id="cdl_mod_id_000000001"
              type="top" />
  </instances>
 </design>
</project>
```

D. EXAMPLE COSSAP DESIGN

This appendix contains a complete, industrially developed Cossap design, being the Decoding of User Data (DUD) component of the UMTS receiver. The structure of the entire UMTS receiver is depicted in Figure D.1, where the DUD component is highlighted and its structure further broken down into sub-modules. As explained in Section 3.3.1, Cossap designs consist of structural/interconnect information stored in `*.v_ent` and `*.v_arc` files which are VHDL-compliant and functional descriptions in `*.gc` files written in GenericC.

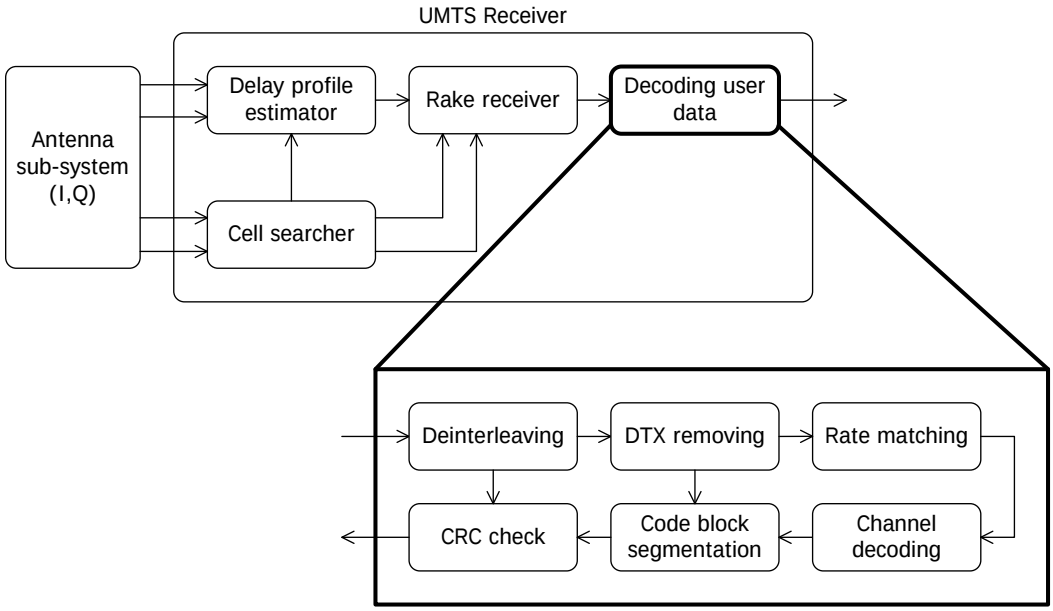

Figure D.1: *Structure of the UMTS receiver and the DUD component*

The **entity** of the DUD component is defined in the `*.v_ent` file, which is made up of three sections. The first section defines the necessary libraries for the definition of the entity. The second section holds the actual entity definition, including the **generic** and **port** statements. Please note that in the interest of brevity, both of these statements have been abridged. The generic statement holds the definitions of all parameters of the entity being defined, while the port statement holds all the port definitions. Finally, the third section of the `*.v_ent` file holds definitions of constants.

```
library COSSAP ;
library LIB_CCODING ;
library LIB_DSP ;
library LIB_MOD63 ;
library LIB_XDISPLAY ;
use COSSAP.COSSAP_TYPES.all ;

entity dud is
  generic(
    StaticFlag                  : in COSSAP_PARAMETER_INTEGER ;
    CompressedFlag              : in COSSAP_PARAMETER_INTEGER ;
      [............PARAMETER DEFINITIONS.............]
    DSP_OUTPUT_RATE_25          : in COSSAP_DSP_OUTPUT_RATE ;
    DSP_OUTPUT_RATE_26          : in COSSAP_DSP_OUTPUT_RATE
  ) ;
  port(
    In_PhCH1                    : in COSSAP_REAL32 ;
    In_PhCH2                    : in COSSAP_REAL32 ;
      [..............PORT DEFINITIONS................]
    Out_TrCH8_crc               : out COSSAP_INTEGER32 ;
    Out_SIR_target              : out COSSAP_REAL32
  ) ;

 constant MODEL_ID : COSSAP_MODEL_ALIAS := "CCODING:dud" ;
 constant TIME_STATISTICS : COSSAP_TIME_STATISTICS := 'F' ;
end dud ;
```

Whereas the outside view of an entity is defined in the `*.v_ent` file, its inside structure, i.e. its **architecture** is defined in the `*.v_arc` file. This file is made up of six sections. The first of these holds definitions of all the components used in the architecture, while the second contains definitions of all the signals in the architecture. The third section binds components used within the architecture to component definitions in the included libraries through use statements. The fourth section defines all the attributes in the system, while the fifth section binds input and output ports of the entity to internal signals. Finally, the sixth section of the `*.v_arc` file contains instantiations of each of the components, including their interconnection to appropriate signals. Please note that all the sections in the following `*.v_arc` example file have been abridged.

```
architecture dud of dud is
  component deintleav_1st_fh3_CCODING
    generic(
      StaticFlag        : in COSSAP_PARAMETER_INTEGER ;
      RunMode           : in COSSAP_PARAMETER_INTEGER ;
      FixedFlag         : in COSSAP_PARAMETER_INTEGER ;
      NormType          : in COSSAP_PARAMETER_INTEGER ;
      WL_TTIOutput      : in COSSAP_PARAMETER_INTEGER ;
      IWL_TTIOutput     : in COSSAP_PARAMETER_INTEGER ;
      WL_Coding         : in COSSAP_PARAMETER_INTEGER ;
      INPUT_SCHEDULE_1  : in COSSAP_INPUT_SCHEDULE ;
      INPUT_SCHEDULE_2  : in COSSAP_INPUT_SCHEDULE ;
      INPUT_SCHEDULE_3  : in COSSAP_INPUT_SCHEDULE ;
      INPUT_SCHEDULE_4  : in COSSAP_INPUT_SCHEDULE ;
      OUTPUT_RATE_1     : in COSSAP_OUTPUT_RATE ;
      OUTPUT_RATE_2     : in COSSAP_OUTPUT_RATE ;
      OUTPUT_RATE_3     : in COSSAP_OUTPUT_RATE ;
      OUTPUT_RATE_4     : in COSSAP_OUTPUT_RATE ;
      DSP_INPUT_RATE_1  : in COSSAP_DSP_INPUT_RATE ;
      DSP_INPUT_RATE_2  : in COSSAP_DSP_INPUT_RATE ;
```

```
    DSP_INPUT_RATE_3  : in COSSAP_DSP_INPUT_RATE ;
    DSP_INPUT_RATE_4  : in COSSAP_DSP_INPUT_RATE ;
    DSP_OUTPUT_RATE_1 : in COSSAP_DSP_OUTPUT_RATE ;
    DSP_OUTPUT_RATE_2 : in COSSAP_DSP_OUTPUT_RATE ;
    DSP_OUTPUT_RATE_3 : in COSSAP_DSP_OUTPUT_RATE ;
    DSP_OUTPUT_RATE_4 : in COSSAP_DSP_OUTPUT_RATE
  ) ;
  port(
    Input_para        : in COSSAP_INTEGER32 ;
    Input_data        : in COSSAP_REAL32 ;
    Input_fixed       : in COSSAP_INTEGER32 ;
    Input_norm        : in COSSAP_REAL32 ;
    Output_para       : out COSSAP_INTEGER32 ;
    Output_data       : out COSSAP_REAL32 ;
    Output_fixed      : out COSSAP_INTEGER32 ;
    Output_norm       : out COSSAP_REAL32
  ) ;
end component ;
component deintleav_2nd_f3_CCODING
  [..........COMPONENT DEFINITION............]
end component ;
component decoding_h3_CCODING
  [..........COMPONENT DEFINITION............]
end component ;
component code_blk_concat_v4_CCODING
  [..........COMPONENT DEFINITION............]
end component ;
component crc_check_v8_CCODING
  [..........COMPONENT DEFINITION............]
end component ;
component dtx_removal_1st_v3_CCODING
  [..........COMPONENT DEFINITION............]
end component ;
component dtx_removal_2nd_v2_CCODING
  [..........COMPONENT DEFINITION............]
end component ;
component derate_match_h2_CCODING
  [..........COMPONENT DEFINITION............]
end component ;

signal SIG_40    : COSSAP_REAL32 ;
signal SIG_76    : COSSAP_REAL32 ;
  [...........SIGNAL DEFINITIONS.............]
signal SIG_1_3   : COSSAP_REAL32 ;
signal SIG_56    : COSSAP_INTEGER32 ;

for M_M33_1   : deintleav_1st_fh3_CCODING
    use entity LIB_CCODING.deintleav_1st_fh3(deintleav_1st_fh3) ;
  [.............USE STATEMENTS..............]
for M_M23_1   : derate_match_h2_CCODING
    use entity LIB_CCODING.derate_match_h2(derate_match_h2) ;

attribute IMPL_TYPE     of M_M33_1 : LABEL is "DEFAULT_IMPLEMENTATION" ;
attribute IMPL_LANGUAGE of M_M33_1 : LABEL is "" ;
attribute IMPL_FILE     of M_M33_1 : LABEL is "" ;
attribute IMPL_XSYMBOL  of M_M33_1 : LABEL is "" ;
  [...........ATTRIBUTE DEFINITIONS..........]
attribute IMPL_TYPE     of M_M23_1 : LABEL is "DEFAULT_IMPLEMENTATION" ;
attribute IMPL_LANGUAGE of M_M23_1 : LABEL is "" ;
attribute IMPL_FILE     of M_M23_1 : LABEL is "" ;
attribute IMPL_XSYMBOL  of M_M23_1 : LABEL is "" ;

begin
```

```
  SIG_14 <= In_PhCH1;
  SIG_15 <= In_PhCH2;
    [........I/O PORT CONNECTIONS.............]
  Out_TrCH8_crc <= SIG_136;
  Out_SIR_target <= SIG_39;

  M_M33_1 : deintleav_1st_fh3_CCODING
    generic map(
      StaticFlag,
      RunMode,
      FixedFlagDIL1,
      NormType,
      WL_TTIOutput,
      IWL_TTIOutput,
      WL_DEC,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1,
      -1
    )
    port map(
      SIG_239_1,
      SIG_162_1,
      SIG_181,
      SIG_182,
      SIG_163,
      SIG_234,
      SIG_186,
      SIG_187
    ) ;
  M_M50_1 : deintleav_2nd_f3_CCODING
    [..........COMPONENT INSTANTIATION.........]
  M_M0_1 : decoding_h3_CCODING
    [..........COMPONENT INSTANTIATION.........]
  M_M156_1 : code_blk_concat_v4_CCODING
    [..........COMPONENT INSTANTIATION.........]
  M_M124_1 : crc_check_v8_CCODING
    [..........COMPONENT INSTANTIATION.........]
  M_M31_1 : dtx_removal_1st_v3_CCODING
    [..........COMPONENT INSTANTIATION.........]
  M_M16_1 : dtx_removal_2nd_v2_CCODING
    [..........COMPONENT INSTANTIATION.........]
  M_M23_1 : derate_match_h2_CCODING
    [..........COMPONENT INSTANTIATION.........]
end dud ;
```

The functionality of each of the sub-modules of the DUD component is described in a separate *.gc file. Hence, this example design contains six GenericC files, corresponding to the six sub-modules of the DUD component (see Figure D.1). Below listed is one of these GenericC file - that corresponding to the Check CRC sub-module.

```
#include <stdio.h>
#include <malloc.h>
#include "ccoding.h"
#include "fixed2float.h"
INPUT_PORT(1)  register int *Input_para;
INPUT_PORT(2)  register int *Input_data;
OUTPUT_PORT(1) register int *Output_para;
OUTPUT_PORT(2) register int *Output_data;
OUTPUT_PORT(3) register int *Output_crc;
PARAMETER(1) int   StaticFlag;
PARAMETER(2) int   nIterations;
PARAMETER(3) int   BTFDFlag;
PARAMETER(4) int   RunMode;
PARAMETER(5) int   FixedFlag;
PARAMETER(6) int   TestFlag;
PARAMETER(7) char* TestFile;
CONST char *InitParaTable[] =
{   "Fmax", "Fi", "I", "Gi_min"
};
CONST char *InParaTable[] =
{   "Aim", "Mim", "Li", "Cim", "Kim",
    "CTi", "CRi", "dNim", "eini", "eplus1",
    "eplus2", "eminus1", "eminus2", "Fixed", "Fi",
    "Hi", "Np_TTIm", "I", "P", "Mode",
    "Sfo", "dummy1", "dummy2", "nCB", "cb",
    "Xcrc", "Xic", "Arest", "dNic", "Gic",
    "Dic", "Qic", "QiTTI",
    "Ntgl", "Nfirst", "TGL", "Np_imaxn", "dPsir", "CMtti"
};
CONST char *OutParaTable[] =
{   "Xim", "Cim", "Kim", "CTi", "CRi",
    "dNim", "eini", "eplus1", "eplus2", "eminus1",
    "eminus2", "Fixed", "Fi", "Hi", "Np_TTIm",
    "I", "P", "Mode", "Sfo", "Mim",
    "Aim", "dummy1", "dummy2", "nCB", "cb",
    "Xic", "dNic", "Gic", "Dic", "Qic",
    "QiTTI", "Ntgl", "Nfirst", "TGL",
    "Np_imaxn"
};
CONST char* InParaFormat  = "%s %d\n";
CONST char* OutParaFormat = "%s %d\n";
CONST char* InDataFormat  = "di: %d\n";
CONST char* OutDataFormat = "do: %d\n";
CONST char* OutCRCFormat  = "%1x\n";
CONST int Rate_Output_para_next = 4;
CONST int Rate_Output_data_next = 0;
CONST int Rate_Output_crc_next = 0;
STATE int Rate_Input_para_next = 4;
STATE int Rate_Input_data_next = 0;
STATE int Rate_Output_para = 0;
STATE int Rate_Output_data = 0;
STATE int Rate_Output_crc= 0;
STATE int Factor;
STATE int Fmax;
STATE int m = 0;
STATE int Ai[Factor];
STATE int Mi[Factor];
STATE int Xi[Factor];
STATE int Li;
STATE int CT;
STATE int Ar = 0;
STATE int Ar_save;
STATE int Xcrc[Factor];
STATE int akku[16];
```

```
STATE int nCB;
STATE int cb;
STATE int ProcMode = INIT_PROC;
STATE FILE* tptr;
STATE FILE* vptr;
RATE(INPUT_PORT(1)) = Rate_Input_para_next;
RATE(INPUT_PORT(2)) = Rate_Input_data_next;
RATE(OUTPUT_PORT(1)) = Rate_Output_para;
RATE(OUTPUT_PORT(2)) = Rate_Output_data;
RATE(OUTPUT_PORT(3)) = Rate_Output_crc;
int crc_dec_forw_proc_v1( int *, const int, const int);
void crc_dec_forw_proc_v2(const int  *,int*,const int,const int);
int floordiv(int, int);
void init_crc_check_v12()
{
  if(TestFlag)
  {
    char *ExpandFile = expandenv( TestFile);
    if( (vptr = fopen( ExpandFile, "w")) == NULL) ExitWithError(1);
    free( ExpandFile);
    fprintf( vptr, "// add header here\n");
    fprintf( vptr, "// CRC check input & output para\n//\n" );
  }
  if(TestFlag)
  {
    char *CRC_Output = "${COSSAP_USER_D}/ccoding/dump/crc_out.${SIMNO}.tp";
    char *ExpandFile = expandenv( CRC_Output);
    if( (tptr = fopen( ExpandFile, "w")) == NULL) ExitWithError(2);
    free( ExpandFile);
    fprintf( tptr, "// add header here\n// CRC output data\n//\n" );
  }
}
void crc_check_v12()
{
  switch(ProcMode)
  {
    case INIT_PROC:
    {
      int dummy = Factor = (StaticFlag) ? Input_para[0]/Input_para[1] : 1 ;
      int *pInput = Input_para;
      register int n;
      int lpara = 16;
      SCRATCH int OutScratch[4];
      SCRATCH int ParaScratch[4*Factor+lpara];
      ccg_Output_para = Output_para = OutScratch;
      ccg_Ai   = Ai   = ParaScratch;
      ccg_Mi   = Mi   = ParaScratch +  Factor;
      ccg_Xi   = Xi   = ParaScratch + 2*Factor;
      ccg_Xcrc = Xcrc = ParaScratch + 3*Factor;
      ccg_akku = akku = ParaScratch + 4*Factor;
      Rate_Output_para = 4;
      if(TestFlag)
      {
        fprintf( vptr, "*** new block: input parameters ***\n");
        Input_para = pInput;
        for( n=0; n<Rate_Input_para_next; n++)
          fprintf( vptr, InParaFormat, InitParaTable[n], *Input_para++);
        fprintf( vptr, "*** new block: output parameters ***\n");
        Output_para = ccg_Output_para;
        for( n=0; n<Rate_Output_para; n++)
          fprintf( vptr, OutParaFormat, InitParaTable[n], *Output_para++);
      }
      Rate_Input_para_next = 33+6*Fmax;
      Rate_Output_para_next = 31+4*Fmax;
```

```
  ProcMode = PARA_PROC;
  break;
}
case PARA_PROC:
{
  int dummy = CT = Input_para[5];
  register int n;
  int *pInput = Input_para;
  int Fi, nCB;
  SCRATCH  int OutScratch[31+4*Fmax];
  ccg_Output_para = Output_para = OutScratch;
  if(CT != TC) nIterations = 1;
  Ai[m] = *Input_para++;
  Mi[m] = *Input_para++;
  Li    = *Input_para++;
  Xi[m] = Mi[m]*(Ai[m]+Li);
  *Output_para++ = Xi[m];
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ =CT=*Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ =Fi=*Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = Mi[m];
  *Output_para++ = Ai[m];
  *Output_para++ = *Input_para++;
  *Output_para++ = *Input_para++;
  *Output_para++ = nCB = *Input_para++;
  if(nCB)
  {
    *Output_para++ = cb = *Input_para++;
    for(n=0;n<nIterations;n++) akku[n] = (cb) ? (akku[n]) : (0);
    Xcrc[m] = *Input_para++;
    *Output_para++ = *Xi = *Input_para++;
    Ar = ( cb) ? Ar_save : 0 ;
    Ar_save = *Input_para++;
    *Output_para++ = *Input_para++;
    *Output_para++ = *Input_para++;
    *Output_para++ = *Input_para++;
    *Output_para++ = *Input_para++;
    *Output_para++ = *Input_para++;
    if(Ar < Ai[m])
      Mi[m] = floordiv(Xcrc[m]+Ar,Ai[m]+Li);
    else
    {
      Mi[m] = 1;
      Mi[m] += floordiv(Xcrc[m]-Li,Ai[m]+Li);
    }
  }
  else
  {
    Input_para += 9; Output_para += 7;
```

```
      Ar = 0; Xcrc[m] =  Xi[m];
      for(n=0;n<nIterations;n++) akku[n] = 0;
    }
    for(n=0; n<Fi; n++)
    {
      *Output_para++ = *Input_para++;
      *Output_para++ = *Input_para++;
      *Output_para++ = *Input_para++;
      *Output_para++ = *Input_para++;
      Input_para += 2;
    }
    if((nIterations > 1) && (nCB > 0))
    {
      printf("Number of iterations must be 1 for VP-mode !\n");
      ExitWithError(970);
    }
    Rate_Output_para = 31+4*Fmax; Rate_Output_data = Rate_Output_crc = 0;
    if(TestFlag)
    {
      int len, k;
      fprintf( vptr, "*** new block: input parameters ***\n");
      Input_para = pInput; len = Rate_Input_para_next -6*Fmax;
      for( n=0; n<len; n++)
        fprintf( vptr, InParaFormat, InParaTable[n], *Input_para++);
      for( k=0; n<Rate_Input_para_next; n++, k=(++k)%6)
        fprintf( vptr, InParaFormat, InParaTable[len+k], *Input_para++);
      fprintf( vptr, "*** new block: output parameters ***\n");
      Output_para = ccg_Output_para; len = Rate_Output_para -4*Fmax;
      for( n=0; n<len; n++)
        fprintf( vptr, OutParaFormat, OutParaTable[n], *Output_para++);
      for( k=0; n<Rate_Output_para; n++, k=(++k)&3)
        fprintf( vptr, OutParaFormat, OutParaTable[len+k], *Output_para++);
    }
    if((!StaticFlag)||(m >= Factor-1))
    {
      Rate_Input_para_next=0;
      Rate_Input_data_next=nIterations*Xcrc[0];
      Rate_Output_para_next=0;
      Rate_Output_data_next=(BTFDFlag&&(Ai[0]==0))?(Mi[0]):(nIterations*(Xcrc[0]-Mi[0]*Li));
      Rate_Output_crc_next=nIterations*Mi[0];
      ProcMode = DATA_PROC;
      if(m) m = 0;
    }
    else
    {
      Rate_Output_para_next = Rate_Output_para;
      m++;
    }
    break;
  }
  case DATA_PROC:
  {
    register int j, k, iter;
    int Aim = Ai[m], btfd_error_flag = 0;
    int Mim = Mi[m], btfd_error = 0;
    int Xcrcm = Xcrc[m], Xcrctmp = Xcrcm;
    int Artmp, Ac;
    int *pInput = Input_data;
    int data_rate_TTI=(CT==TC)?(nIterations*(Xcrc[0]-Mi[0]*Li)):Xcrc[0]-Mi[0]*Li;
    int data_out_rate = 0;
    int crc_out_rate = 0;
    SCRATCH int OutScratch_data[(BTFDFlag&&(Aim==0))?(Mim):(data_rate_TTI)];
    SCRATCH int OutScratch_crc [(CT==TC)?(nIterations*Mim):(Mim)];
    ccg_Output_data = Output_data = OutScratch_data;
```

```
ccg_Output_crc  = Output_crc  = OutScratch_crc;
if(CT != TC)
  nIterations = 1;
if(FIXED_MODE( RunMode, FixedFlag ))
{
  if(Li == 0)
  {
    for(iter=0; iter<nIterations; iter++)
    {
      Xcrctmp = Xcrcm;
      Artmp = Ar;
      while( Xcrctmp > 0 )
      {
        Ac = Aim - Artmp;
        if( Ac <= Xcrctmp )
        {
          for(j=0;j<Ac;j++)
            *Output_data++ = *Input_data++;
          *Output_crc++ = 0;
          Xcrctmp -= Ac;
          Artmp = 0;
          crc_out_rate++; data_out_rate += Ac;
        }
        else
        {
          for(j=0;j<Xcrctmp;j++)
            *Output_data++ = *Input_data++;
          data_out_rate += Xcrctmp;
          Xcrctmp = 0;
        }
      }
    }
  }
  else
  {
    register int *Output = Output_data;
    int Aim_tmp;
    if(BTFDFlag&&((*Input_data)>>1))
    {
      if(cb>0)
      {
        printf("crc_check_v12: Cannot have multiple codeblocks during BTFD !\n");
        ExitWithError(971);
      }
      btfd_error = *Input_data;
      *Input_data = (*Input_data > 0) ? 0 : 1 ;
    }
    for(iter=0; iter<nIterations; iter++)
    {
      Xcrctmp = Xcrcm;
      Artmp = Ar;
      while(Xcrctmp>0)
      {
        Ac = Aim - Artmp;
        if(Ac<=0)
        {
          crc_dec_forw_proc_v2( Input_data, &akku[iter], 0, Li );
          *Output_crc++ = (akku[iter]) ? (1) : (0);
          Input_data += Li;
          Xcrctmp -=  Li;
          Artmp = 0;
          akku[iter] = 0;
          crc_out_rate++;
        }
```

```
else
{
  if(Ac+Li<=Xcrctmp)
  {
    crc_dec_forw_proc_v2(Input_data,&akku[iter],Ac,Li);
    if((btfd_error==BTFD_ERROR)||(btfd_error==-BTFD_ERROR))
    {
      *Output_crc++ = 0;
      btfd_error_flag = 1;
    }
    else
    {
      if((btfd_error==BTFD_ONLYBLER_ERROR)||(btfd_error==-BTFD_ONLYBLER_ERROR))
      {
        *Output_crc++ = 1;
        btfd_error_flag = 1;
      }
      else
      {
        *Output_crc++ = (akku[iter]) ? (1) : (0);
      }
    }
    for(j=0;j<Ac;j++)
      *Output_data++ = *Input_data++;
    Input_data += Li;
    Xcrctmp -=  (Ac + Li);
    Artmp = 0;
    akku[iter] = 0;
    crc_out_rate++;
    if(BTFDFlag&&(Aim==0))
    {
      *Output = btfd_error;
      Output += 1;
      data_out_rate++;
    }
    else
    {
      if(btfd_error)
      {
        *Output = btfd_error;
        Output += Ac;
      }
      data_out_rate += Ac;
    }
  }
  else
  {
    if(Ac<=Xcrctmp)
    {
      crc_dec_forw_proc_v2( Input_data, &akku[iter], Ac, -Li );
      for(j=0;j<Ac;j++)
        *Output_data++ = *Input_data++;
      Xcrctmp -=  Ac;
      Artmp = 0;
      data_out_rate += Ac;
    }
    else
    {
      crc_dec_forw_proc_v2( Input_data, &akku[iter], Xcrctmp, -Li );
      for(j=0;j<Xcrctmp;j++)
        *Output_data++ = *Input_data++;
      Artmp += Xcrctmp;
      data_out_rate += Xcrctmp;
      Xcrctmp = 0;
```

```
            }
          }
        }
      }
    }
  }
  if((crc_out_rate!=(nIterations*Mim))
    ||(data_out_rate!=((BTFDFlag&&(Aim == 0))?(Mim):(nIterations*(Xcrcm-Mim*Li)))))
  {
    printf("crc_check_v12: Something is wrong with the output rates!\n");
    ExitWithError(972);
  }
}
else
{
  if(Li==0)
  {
    for(iter=0;iter<nIterations;iter++)
    {
      for(k=0;k<Mim;k++)
      {
        for(j=0;j<Aim;j++)
          *Output_data++ = *Input_data++;
        *Output_crc++ = 0 ;
      }
    }
  }
  else
  {
    register int *Output = Output_data;
    int Aim_tmp;
    if(BTFDFlag&&((*Input_data)>>1))
    {
      btfd_error = *Input_data;
      *Input_data = (*Input_data > 0) ? 0 : 1 ;
    }
    for(iter=0;iter<nIterations;iter++)
    {
      for(k=0;k<Mim;k++)
      {
        if( (btfd_error==BTFD_ERROR)||(btfd_error==-BTFD_ERROR))
        {
          *Output_crc++ = 0;
          btfd_error_flag = 1;
        }
        else
        {
          if((btfd_error==BTFD_ONLYBLER_ERROR)||(btfd_error==-BTFD_ONLYBLER_ERROR))
          {
            *Output_crc++ = 1;
            btfd_error_flag = 1;
          }
          else
          {
            *Output_crc++ =crc_dec_forw_proc_v1(Input_data,Aim,Li);
          }
        }
        for(j=0;j<Aim;j++)
          *Output_data++ = *Input_data++;
        Aim_tmp=(BTFDFlag&&(Aim==0)) ? 1 : Aim ;
        if( BTFDFlag && (Aim == 0))
        {
          *Output = btfd_error;
          Output += 1;
```

```
            }
            else if(btfd_error)
            {
              *Output = btfd_error;
              Output += Aim;
            }
            Input_data  += Li;
          }
        }
      }
    }
    Rate_Output_para = 0;
    Rate_Output_data=(BTFDFlag&&!Aim) ? Mim : nIterations*(Xcrcm-Mim*Li);
    Rate_Output_crc=nIterations*Mim;
    Input_data = pInput;
    for(k=TestFlag>1?Rate_Input_data_next:0;k-->0;)
      fprintf(vptr,InDataFormat,*Input_data++);
    Output_data = ccg_Output_data;
    for( k=TestFlag>1?Rate_Output_data:0;k-->0;)
      fprintf(vptr,OutDataFormat,*Output_data++);
    Output_crc = ccg_Output_crc;
    for( k=TestFlag?Rate_Output_crc:0 ;k-->0;)
      fprintf(tptr,OutCRCFormat,*Output_crc++);
    if(!StaticFlag)
    {
      Rate_Input_para_next=33+6*Fmax; Rate_Input_data_next = 0;
      Rate_Output_para_next = 31+4*Fmax;
      Rate_Output_data_next = Rate_Output_crc_next = 0;
      ProcMode = PARA_PROC;
    }
    else if(Factor>1)
    {
      if(++m>=Factor)
        m=0;
      Rate_Input_data_next=nIterations*Xcrc[m];
      Rate_Output_data_next=(BTFDFlag&&!Ai[m])?Mi[m]:nIterations*(Xcrc[m]-Mi[m]*Li);
      Rate_Output_crc_next=nIterations*Mi[m];
    }
   }
  }
  SetInputSchedule(1)=Rate_Input_para_next;
  SetInputSchedule(2)=Rate_Input_data_next;
}
void post_crc_check_v12()
{
  if(TestFlag)
  {
    fprintf(tptr,"// EOF\n");fclose(tptr);
    fprintf(vptr,"// EOF\n");fclose(vptr);
  }
}
```

E. OPTIMIZATION RESULTS FOR THE MIMO DESIGN

This appendix contains detailed results of applying the three optimization methods of the *fixify* environment on the MIMO receiver design, as discussed in Section 4.4. Table E.1 shows the optimization results obtained by the restricted-set full search algorithm, Table E.2 the greedy algorithm, and Table E.3 the branch-and-bound algorithm.

SQNR$_{lim}$ (dB)	E_{lim}	c	E_{max_tot}	SQNR$_{min}$ (dB)	Runtime (hh:mm:ss)
5	0,56234133	96	0,13975700	17,09	04:50:27
10	0,31622777	96	0,13975700	17,09	04:47:02
15	0,17782794	96	0,13975700	17,09	04:49:40
20	0,10000000	112	0,07365257	22,66	04:46:56
25	0,05623413	128	0,02288618	32,81	04:49:24
30	0,03162278	128	0,02288618	32,81	04:46:48
35	0,01778279	152	0,01307150	37,36	04:46:07
40	0,01000000	160	0,00712441	42,95	04:50:51
45	0,00562341	192	0,00008121	81,81	04:47:53
50	0,00316228	192	0,00008121	81,81	04:49:08
55	0,00177828	192	0,00008121	81,81	04:48:19
60	0,00100000	192	0,00008121	81,81	04:49:43
65	0,00056234	192	0,00008121	81,81	04:49:18
70	0,00031623	192	0,00008121	81,81	04:48:43
75	0,00017783	192	0,00008121	81,81	04:48:45
80	0,00010000	192	0,00008121	81,81	04:45:56
85	0,00005623	208	0,00005479	85,23	05:00:39
90	0,00003162	224	0,00003130	90,09	04:52:59
95	0,00001778	256	0,00000429	107,34	04:52:43
100	1,00001000	256	0,00000429	107,34	04:53:44
				Runtime summary	
				Min	04:45:56
				Avg	04:49:45
				Max	05:00:39

Table E.1: *Full search (8,16,32) optimization results*

The column SQNR$_{lim}$ represents the lower bound on the acceptable SQNR, as given

by the designer. In these experiments, the range of SQNR [5db,100db] is covered in 5db steps. The corresponding E_{lim} values (though Equation 4.1) are shown in the second column. The third column shows the resulting values of the implementation cost c. The fourth column shows E_{max_tot}, the maximal total error for the resulting fixed-point configuration (as described in Section 4.2.4), while the fifth column shows the corresponding SQNR value (though Equation 4.1), i.e. the worst-case SQNR for the resulting fixed-point configuration. Finally, in the sixth column, the required runtime for the optimization is given in hours, minutes, and seconds. At the bottom of each table, the summary of runtime statistics is shown, indicating the minimum, average, and maximum runtime for the particular optimization.

SQNR$_{lim}$ (dB)	E_{lim}	c	E_{max_tot}	SQNR$_{min}$ (dB)	Runtime (hh:mm:ss)
5	0,56234133	63	0,54527452	5,27	00:02:00
10	0,31622777	70	0,27468101	11,22	00:02:00
15	0,17782794	74	0,16485223	15,66	00:01:55
20	0,10000000	81	0,08568203	21,34	00:02:02
25	0,05623413	95	0,05600756	25,04	00:02:00
30	0,03162278	99	0,03039700	30,34	00:02:00
35	0,01778279	103	0,01766891	35,06	00:01:55
40	0,01000000	106	0,00964394	40,31	00:02:00
45	0,00562341	116	0,00501345	46,00	00:02:02
50	0,00316228	125	0,00270398	51,36	00:01:55
55	0,00177828	136	0,00177425	55,02	00:01:53
60	0,00100000	137	0,00099635	60,03	00:02:00
65	0,00056234	144	0,00056041	65,03	00:01:58
70	0,00031623	150	0,00030150	70,41	00:01:52
75	0,00017783	158	0,00015566	76,16	00:02:00
80	0,00010000	165	0,00009980	80,02	00:02:00
85	0,00005623	173	0,00005524	85,16	00:01:58
90	0,00003162	180	0,00003162	90,00	00:01:55
95	0,00001778	183	0,00001698	95,40	00:02:00
100	1,00001000	187	0,00000991	100,08	00:01:55
				Runtime summary	
				Min	00:01:52
				Avg	00:01:58
				Max	00:02:02

Table E.2: *Greedy optimization results*

SQNR$_{lim}$ (dB)	E_{lim}	c	E_{max_tot}	SQNR$_{min}$ (dB)	Runtime (hh:mm:ss)
5	0,56234133	63	0,54527452	5,27	00:02:20
10	0,31622777	69	0,30788842	10,23	00:02:28
15	0,17782794	74	0,16485223	15,66	00:02:28
20	0,10000000	81	0,08568203	21,34	00:02:20
25	0,05623413	92	0,05370655	25,40	00:02:47
30	0,03162278	98	0,02648132	31,54	00:02:57
35	0,01778279	103	0,01766891	35,06	00:02:28
40	0,01000000	106	0,00964394	40,31	00:02:26
45	0,00562341	116	0,00501345	46,00	00:02:22
50	0,00316228	125	0,00270398	51,36	00:02:15
55	0,00177828	131	0,00173927	55,19	00:02:55
60	0,00100000	137	0,00099635	60,03	00:02:39
65	0,00056234	144	0,00056041	65,03	00:02:32
70	0,00031623	150	0,00030150	70,41	00:02:16
75	0,00017783	158	0,00015566	76,16	00:02:26
80	0,00010000	165	0,00009980	80,02	00:02:14
85	0,00005623	170	0,00005568	85,09	00:02:30
90	0,00003162	175	0,00002643	91,56	00:02:28
95	0,00001778	180	0,00001624	95,79	00:02:36
100	1,00001000	187	0,00000991	100,08	00:02:28
				Runtime summary	
				Min	00:02:14
				Avg	00:02:30
				Max	00:02:57

Table E.3: *Branch-and-bound optimization results*