Automated Verification Pattern Refinement for Virtual Prototypes

Pavle Belanović, Martin Holzer, Bastian Knerr, and Markus Rupp
Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms
Vienna University of Technology, Vienna, Austria
pbelanov@nt.tuwien.ac.at

***Abstract*— Extreme technological and commercial pressures on the design of embedded systems for wireless communications create the need for new design techniques with improved efficiency and higher levels of reuse. Automation of the verification pattern refinement process is one of the most promising candidates for achieving these improvements. An integrated environment for automated verification pattern refinement from the algorithmic to the virtual prototype abstraction level is presented, showing acceleration of the design process, improved reuse, as well as better quality through elimination of manual coding errors.**

I. Introduction

Design of modern embedded systems experiences high pressures from both technical and commercial aspects. On the one hand, ever-increasing requirements for power and silicon area efficiency, as well as rampantly increasing demand for performance, make the design of embedded systems one of the most technically challenging tasks today. On the other hand, the extremely competitive global market for consumer electronics unabatedly shrinks the time-to-market for most products. Especially high pressures are experienced in the wireless communications sector, where unprecedented rates of advancement are experienced in form-factor reduction, increased performance with new protocols and services, and demand for extreme power efficiency.

These factors have led to the appearance of the so-called design gap [1], [2], where the industry's ability to design has been outstripped by the availability of underlying silicon resources [3] and by the sheer demand for increased performance. The design gap highlights the growing problem of the need for new, more efficient design techniques to meet the said technological and commercial challenges [4], [5].

This challenge affects both the design and verification flows. Indeed, verification is now seen as the true bottleneck of the overall design process, currently accounting for 50%-80% of the total design effort [6]. Hence, one of the best opportunities for closing the design gap lies in raising the efficiency of the verification flow. As will be shown, automation of the verification pattern refinement process leads to increased reuse in the design process, leading to improved efficiency, as well as to elimination of the human coding errors, leading in turn to improved quality. We present an integrated environment for automated verification pattern refinement between algorithmic and virtual prototype (VP) abstraction levels.

The rest of this paper is organized as follows. Section I-A presents the concept of verification pattern refinement, while Section I-B surveys the existing approaches to automating this process. Section I-C introduces the concept of virtual prototyping, an existing environment for development of VPs, and how this environment has been extended into the verification flow. In Section I-D, the model of the hardware platform assumed in this work is given. The environment for automated verification pattern refinement is presented in Section II, including discussions on the verification at both algorithmic and VP levels, the formal interface specification, and the Test Generator Script (TGS), the tool which automates the actual refinement process. Finally, the conclusions of the paper are summarized in Section III.

A. Verification Pattern Refinement

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.

As the model of the design progresses from one refinement level to another, it needs to be verified for correct functionality at each level. Hence, the model of the system at each refinement level has associated with it a set of verification patterns, designed to verify correct functionality of the corresponding model.

The verification patterns at each new level in the design flow are traditionally created from the verification patterns at the previous refinement level. This is shown in Figure 1. We refer to this process henceforth as *verification pattern refinement*.

Whereas a great multitude of EDA tools and research work exists for automating refinement of system models between all the various refinement levels, there is a distinct lack of such support for verification pattern refinement. 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

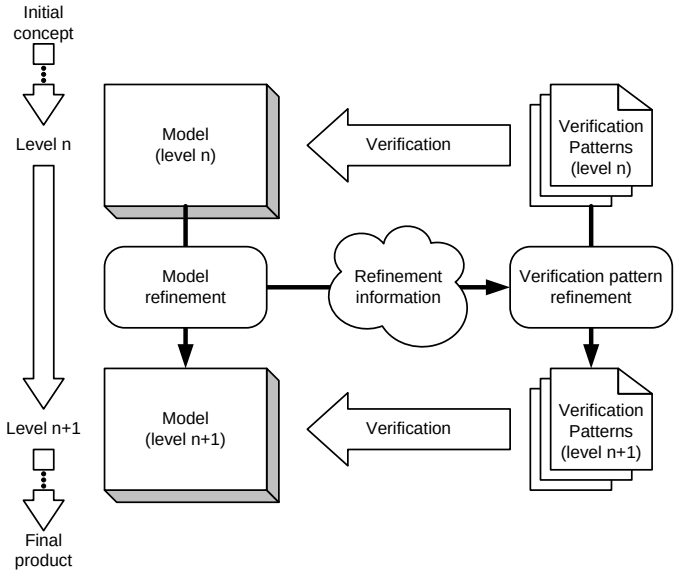

Fig. 1. Conceptual view of parallel refinement of the model and the associated verification patterns

model refinement, to produce the new verification patterns (see Figure 1). Hence, two distinct tasks can be recognised in the process of verification pattern refinement:

- **Reformatting** of verification pattern data, to fit the new format required at the next refinement level.
- **Enrichment** of the same data, with the refinement information (see Figure 1), which does not appear in the original verification patterns, but is a necessary component in the newly created verification patterns.

Although the reformatting task can be, and frequently is, fully automated, current approaches to verification pattern refinement require manual effort from the designer in order to complete the enrichment task, for which traditionally no formal framework exists. Automating the process of verification pattern refinement would significantly accelerate the design process, increase reuse, and eliminate manual coding errors unavoidably created by designers, thus increasing quality. However, automation of this process requires the establishment of a formal framework for refinement of verification patterns.

B. Related Work

Several previous research efforts, both academic and industrial, to automate the verification pattern refinement process exist. Varma et al. [7] present an approach to reusing pre-existing verification 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 verification vectors. However, this approach is applicable only to hardware virtual components.

On the other hand, Stöhr et al. [8] 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 verification 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.

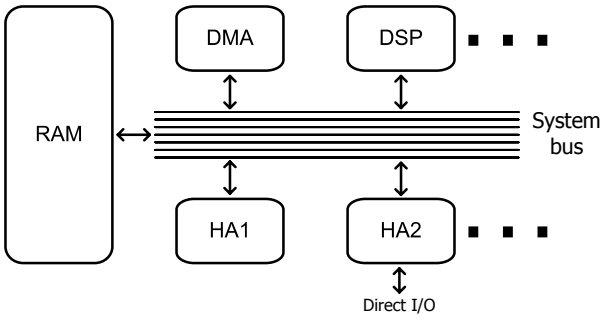

Fig. 3. Target hardware platform

An automated testing framework offered by Odin Technologies called Axe [9] 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 environment for automated verification pattern refinement presented here 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 VP abstraction levels.

C. Virtual Prototyping

One of the most useful techniques for acceleration of the design process is virtual prototyping [10]–[13]. This technique resolves interface dependencies between all system components, thus enabling parallel, rather than sequential, development of hardware and software components of the system, as shown in Figure 2. Additional gains in efficiency and quality can be achieved through automated generation of virtual prototypes directly from algorithmic models [14].

Significant further acceleration of the design process, increase in reuse, and elimination of manual coding errors can be achieved by automating the corresponding verification flow, from algorithmic models to virtual prototypes. Here we present an integrated environment for automated verification pattern refinement between these abstraction levels.

D. Model of Hardware Platform

The structure of the hardware platform assumed in this work is a generic multi-processor system-on-chip (SoC) architecture. At least one processor core, such as the StarCore DSP [15], is present in the architecture, as shown in Figure 3.

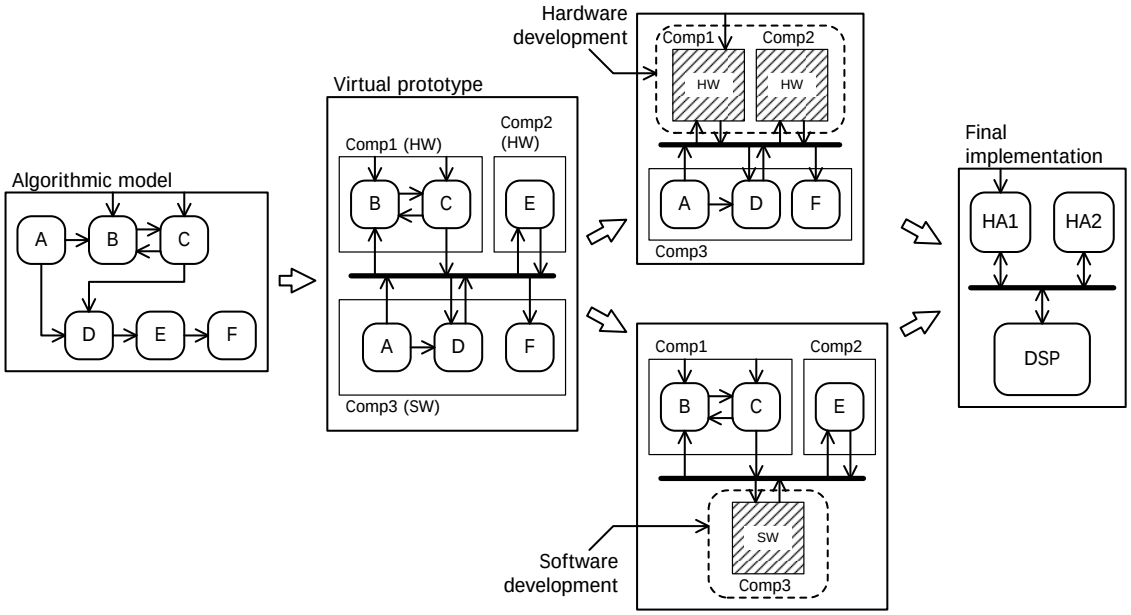

Fig. 2. System development using a virtual prototype

All the system components assigned to software implementation will be targeted to one of these processor cores. Also present in the system are a number of hardware accelerator (HA) blocks. These contain custom silicon designs, to provide accelerated processing for time-critical system functions. All the system components assigned to hardware implementation will be realised as these HA blocks. The system also contains one or more banks of system memory and a dedicated direct memory access (DMA) controller, serving the processor cores as well as the HA blocks.

Communication on this hardware platform is facilitated by at least one system bus, such as an AMBA bus for example, connecting all system components. Additionally, HA blocks may be provided with dedicated direct I/O ports, for off-chip communications.

II. Environment

The presented environment for automated verification pattern refinement creates VP level verification patterns from those at the algorithmic level. Hence, this environment performs both of the required steps, *reformatting* and *enrichment*, automatically. The structure of the environment is shown in Figure 4.

The algorithmic level verification patterns, described in Section II-A, are the inputs into the environment, together with the formal interface specification, which is described in Section II-C. The outputs of the environment are VP level verification patterns, in the form of a memory image, as described in Section II-B. Section II-D describes the Test Generator Script (TGS), the tool which performs both the formatting and the enrichment steps of the verification pattern refinement.

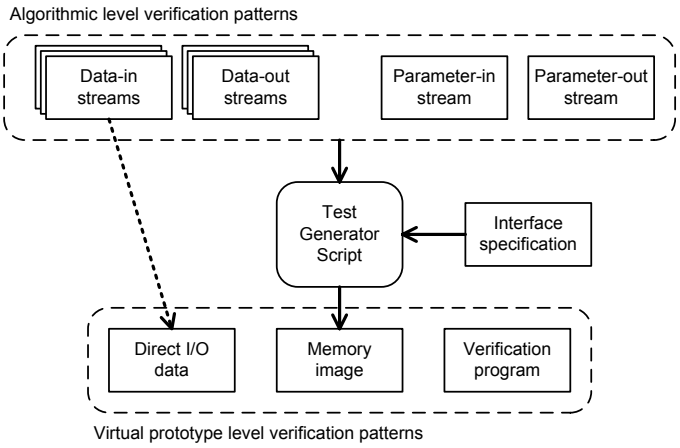

Fig. 4. Structure of the environment for automatic verification pattern refinement

A. Algorithm Verification

At the algorithmic level, the model of the system contains no architectural information and the partitioning of the system is done 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 communication 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 all those going out of the model, also exist. The complete set of streams is shown as algorithmic-level verification patterns in Figure 4.

Since no architectural or implementation information is yet known 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. Thus, no absolute timing is present in the COSSAP streams. Synchronisation among streams is achieved through their block-based structure, which allows the availability of data in all data channels to be determined in a relative, rather than absolute sense.

B. VP Verification

Verification at the virtual prototype level requires the following:

- Device Under Verification (DUV)
- Verification patterns
- Verification program (runs on the CPU, applies the verification patterns to the DUV)

In this work, we focus on verification of system components assigned to hardware implementation, because of the increased complexity of their verification. Verification of software components is entirely analogous, but has reduced complexity, because no off-CPU communication (over the system bus or direct) is involved.

It is important to note that the system architecture (see Figure 3) enforces the separation of verification patterns into two types, according to how they are communicated to the DUV. There exist verification patterns communicated to the DUV through the system bus (stored in a structured memory image) and those communicated to the DUV through its direct I/O interfaces (supplied directly to the DUV during functional simulation). Both of these types of verification patterns are shown as VP level verification patterns in Figure 4, together with the necessary verification program.

The verification program runs on one of the processor cores 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.

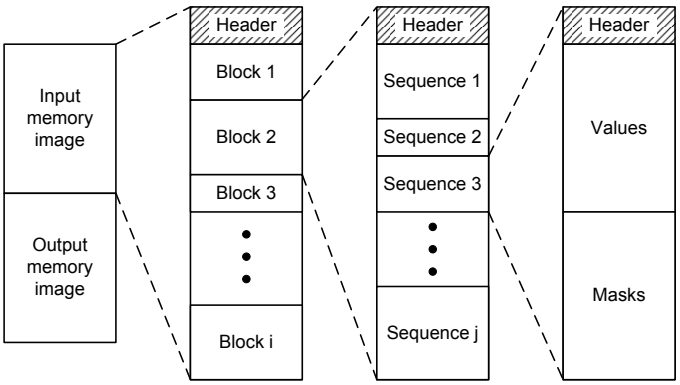

Fig. 5. Structure of the memory image

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

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 5. It is primarily 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) as $n = (\bar{m} \cdot c) + (m \cdot v)$, or a simple 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.

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) as $t = m \cdot (c \oplus v)$, 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.

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

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 models of the CPU and the system bus compliant to the VSIA standard [16], are also accurate to this time resolution within the functional simulation of the complete system.

C. Interface Specification

The interface specification (see Figure 4) 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 1, 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. Indeed, the formal representation of the interface specification and its use in the automation of the enrichment phase of verification pattern refinement is the unique contribution of this environment.

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, together with their bit-exact addresses in the register space 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.

D. 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 4. Its main function is to create the VP level verification patterns, i.e. perform both steps in the verification pattern refinement process automatically.

In order to achieve this, the TGS creates the structure of the memory image as shown in Figure 5. The reformatting step of the verification pattern refinement process is achieved by interleaving the block-based structure of the algorithmic verification patterns, followed by the analysis of the resulting single stream of patterns. As a result of this analysis, the structure of the memory image, with associated block, sequence, and pointer structures can be created.

The second step in the verification pattern refinement process is the enrichment of the verification patterns with refinement information, i.e. architectural details. This task achieves the filling out of the empty memory image structure with the actual verification pattern values, with correct bus interface formats, including appropriate register mapping. Hence, in order to complete this task, the TGS constructs each sequence of each block, both in the input and the output memory image, by bitwise combination of the algorithmic verification patterns, according to the register mapping found in the interface specification. Also, the TGS creates the appropriate bitwise masks found in each sequence, again from the 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.

III. Conclusion and Future Work

The presented environment for automated verification pattern refinement demonstrates the reuse of verification patterns from the algorithmic to the VP abstraction levels. The automation of this otherwise manual process accelerates the design process, as well as reduces manual coding errors, leading to improved quality.

The application of the presented environment is limited in its general applicability in several aspects. Firstly, the algorithmic descriptions considered in this work come from the COSSAP environment. While system descriptions originating in any of the numerous other environments for algorithmic modelling have not as yet been considered, the modular nature of the presented environment offers the possibility to process these other types of descriptions as well, with minimal modifications and/or extensions. In particular, processing algorithmic descriptions in SystemC is being considered as a future extension to the presented environment, due to the strong presence of SystemC in the EDA market [17]–[19]. This will require only minimal extension to the presented environment, due to the already present ability of the underlying framework to process algorithmic descriptions in SystemC [20].

Furthermore, the verification strategy presented here has been implemented only for systems built around the StarCore DSP [15]. However, the modular nature of the verification environment ensures the applicability of the environment to systems built both around other processor cores as well as multiprocessor systems, with only minimal modifications. One of the directions of future work being considered includes extending the environment to systems using other processor cores, by creating verification programs for a set of supported cores. This may also require reformatting the associated memory images, to accommodate varying register and memory widths. However, since these widths are parameters in the TGS, no further modification to this script itself is required in order to adopt it to any set of processor cores.

Acknowledgment

This work has been funded by the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms. The authors would like to acknowledge the ongoing co-operation with Infineon Technologies and in particular thank Guillaume Sauzon, Thomas Herndl, Ahmad Sarashgi, Wolfgang Haas, and Johann Glaser for their collaboration.

References

[1] 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.

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

[3] G.E.Moore, "Cramming More Components Onto Integrated Circuits," *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, April 1965.

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

[5] 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*, 2005.

[6] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-Chip Designs*. Kluwer Academic Publishers, 1998.

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

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

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

[10] R. Ernst, "Codesign of Embedded Systems: Status and Trends," *IEEE Design and Test of Computers*, pp. 45–54, 1998.

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

[12] C. Valderrama, "Virtual Prototyping For Modular And Flexible Hardware-Software Systems," *Design Automation for Embedded Systems Journal*, vol. 2, no. 2, pp. 267–282, 1997.

[13] N. S. Voros, L. Sánchez, A. Alonso, A. N. Birbas, M. Birbas, and A. Jerraya, "Hardware/Software Co-Design of Complex Embedded Systems," *Design Automation for Embedded Systems*, vol. 8, pp. 5–49, 2003.

[14] 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*, June 2004, pp. 114–118.

[15] StarCore DSP, www.starcore-dsp.com.

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

[17] Open SystemC Initiative, www.systemc.org.

[18] CoWare, Inc., "SoC Platform-Based Design Using ConvergenSC/SystemC," July 2002, www.coware.com.

[19] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[20] 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*, July 2003, pp. 288–291.

[21] N. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2003.