Hindawi Publishing Corporation
EURASIP Journal on Applied Signal Processing
Volume 2006, Article ID 32408, Pages 1–12
DOI 10.1155/ASP/2006/32408

A Fully Automated Environment for Verification of Virtual Prototypes

P. Belanović, B. Knerr, M. Holzer, and M. Rupp

Institute of Communications and Radio Frequency Engineering, Vienna University of Technology, 1040 Vienna, Austria

Received 15 October 2004; Revised 29 March 2005; Accepted 25 May 2005

The extremely dynamic and competitive nature of the wireless communication systems market demands ever shorter times to market for new products. Virtual prototyping has emerged as one of the most promising techniques to offer the required time savings and resulting increases in design efficiency. A fully automated environment for development of virtual prototypes is presented here, offering maximal efficiency gains, and supporting both design and verification flows, from the algorithmic model to the virtual prototype. The environment employs automated verification pattern refinement to achieve increased reuse in the design process, as well as increased quality by reducing human coding errors.

1. INTRODUCTION

Complexity of modern embedded systems, particularly in the wireless communications domain, grows at an astounding rate. This rate is so high that the algorithmic complexity now significantly outpaces the growth in complexity of underlying silicon implementations, which proceeds according to the famous Moore's Law [1]. Furthermore, algorithmic complexity even more rapidly outpaces design productivity, expressed as the average number of transistors designed per staff/month [2, 3]. In other words, current approaches to embedded system design are proving inadequate in the struggle to keep up with system complexity.

Hence, a number of new system design techniques with potential to speed up design productivity are intensively researched [4, 5]. One of these techniques known as virtual prototyping [6–8] speeds up the design process by enabling development of hardware and software components of the embedded system in parallel.

Development of a comprehensive design environment for automatic generation and verification of virtual prototypes (VPs) from an algorithmic-level description of the system is presented here. Section 1.1 describes the concept of a VP in closer detail and Section 1.2 explains the model of the hardware platform used in this work. A survey of related work, including a comparison of the presented environment with the most advanced current approaches, is given in Section 1.3. The design environment for automatic generation of VPs is described in detail in Section 2. The part of the presented environment concerned with automated verification pattern refinement for VPs is presented in Section 3, together with an example design. Finally, conclusions are drawn in Section 4.

1.1. Virtual prototype concept

System descriptions at algorithmic level contain no specific implementation details. Hence, before implementation of the system can begin, the algorithmic description is partitioned, that is, each component in the description is assigned to 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 1, top chart).

Virtual prototyping [9] is a technique which can eliminate most of this penalty and thus dramatically shorten the development cycle. A VP is a 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.

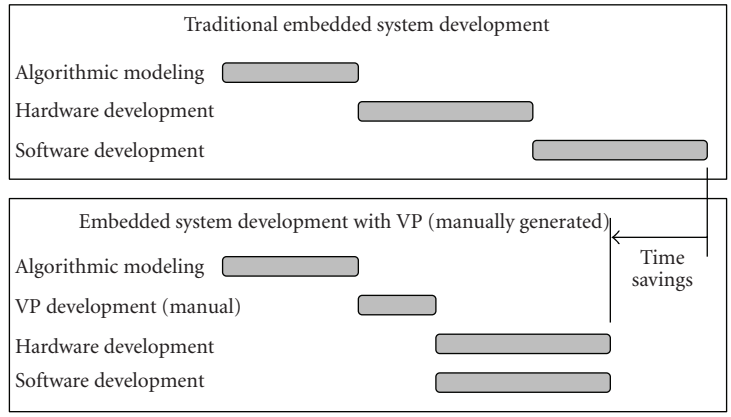

Figure 1: Shortening of the design cycle by the VP technique.

The speedup in the system development cycle by employing virtual prototyping is achieved as depicted in Figure 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 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. Thus, the VP of the system is created.

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 in the system architecture. In Figure 2, blocks B and C form the VP component 1, whereas block E alone forms the VP component 2.

Development of the hardware implementation of VP component 1 is done against the hardware-software interface defined in the VP. Similarly, the software implementation of VP component 2 relies on the existence of the same hardware-software interface. At the same time, the development of the software implementation of VP component 3 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 1 (bottom chart), this allows the time savings which make VP a desirable design technique.

1.2. Model of hardware platform

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

Communications on this hardware platform are 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.

1.3. Related work

Extension of the virtual prototyping environment into the verification flow requires automated verification pattern refinement, as explained in Section 3. Several previous research efforts in this area exist. Varma and Bhatia [10] present an approach to reusing preexisting verification programs for virtual components. This approach includes a fully automated reuse 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.

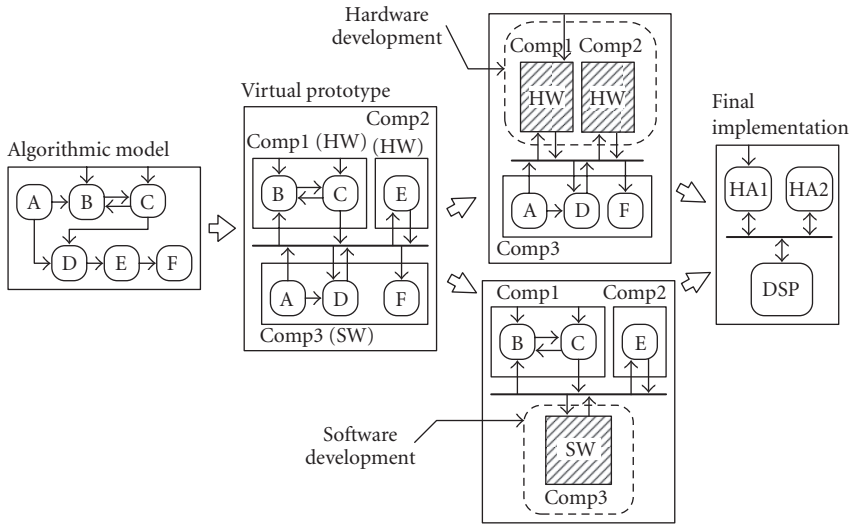

Figure 2: System development using a VP.

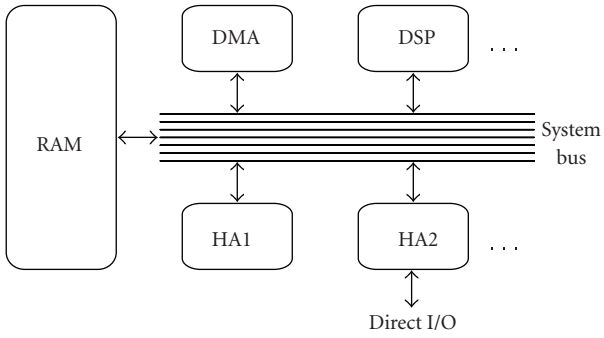

Figure 3: Target hardware platform.

On the other hand, Stöhr et al. [11] present FlexBench, a fully automated methodology for reuse of component-level stimuli in system verification. While this environment presents a novel structure which supports verification pattern reuse 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.

An automated testing framework offered by Odin Technologies called Axe [12] also offers automated reuse 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.

2. AUTOMATED VIRTUAL PROTOTYPE GENERATION

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, 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 [13, 14]. This is illustrated in Figure 4.

The automatic VP generation environment presented here is depicted in Figure 5. 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 labelling 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.

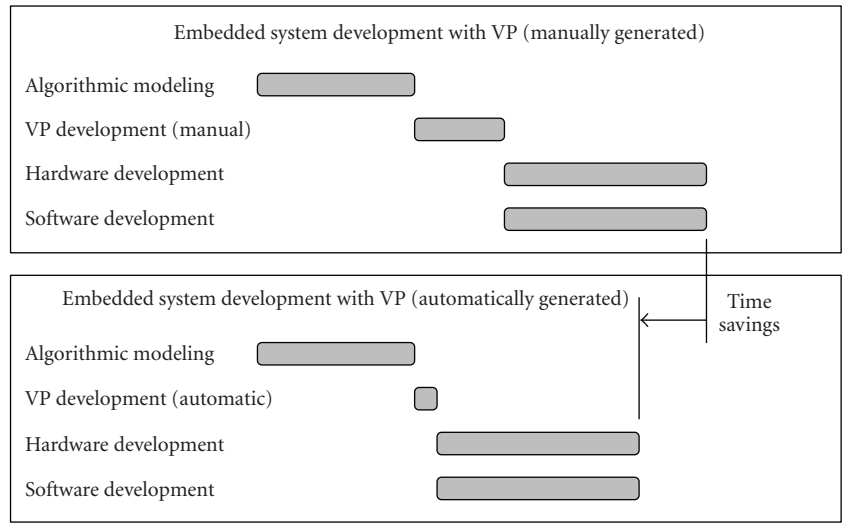

Figure 4: Shortening of the design cycle by automating VP generation.

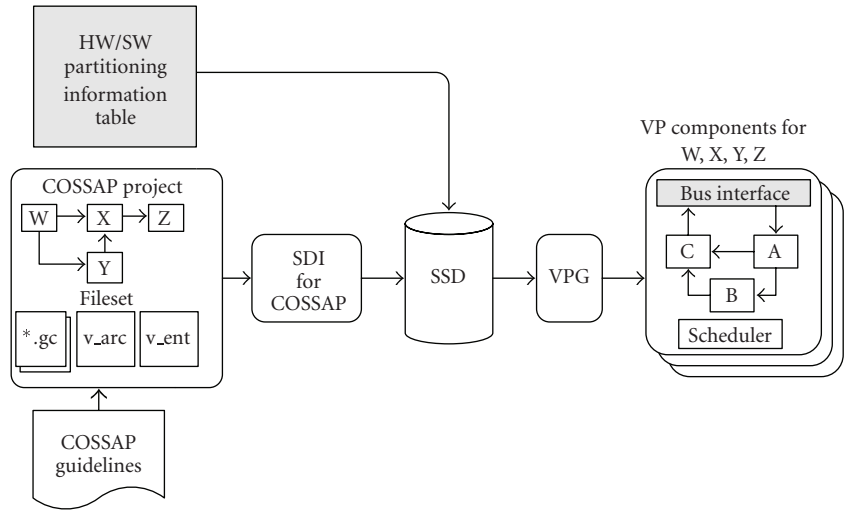

Figure 5: Design environment for automatic generation of VPs.

2.1. 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 modelling 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 and is read into the SSD by the system description interface (SDI). 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 (extension to ANSI C proprietary to the COSSAP environment) and has to be formatted in accordance with specific guidelines. These guidelines ensure compatibility of the GenericC code with tools in the second phase of the automatic VP generation. Suitably formatted functional component descriptions are placed directly into the SSD.

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. Also, possibilities for automated hardware/software partitioning exist and have been successfully applied to the presented environment [15], 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.

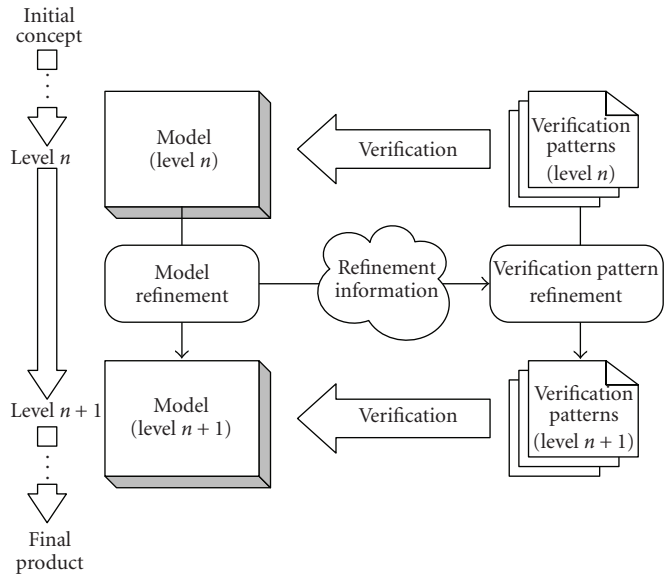

Figure 6: Conceptual view of parallel refinement of the model and the associated verification patterns.

2.2. Virtual prototype generation

A VP component is composed of several parts, as shown in Figure 5. The core of the VP component is the recreated interconnected block structure, as found in the algorithmic-level model—blocks A, B, and C in Figure 5. Additionally, the VP component contains a scheduler which controls the execution of each block, according to the current input and output sample rates of each block and the availability of data to be processed. 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 5, because it needs to be created manually, depending on the bus type, communications protocol, and processor core(s) used in the system.

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 with the VSIA standard [16] 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. 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.

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 6. We refer to this process henceforth as *verification pattern refinement*.

Whereas a great multitude of EDA tools and reseach 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 model refinement, to produce the new verification patterns (see Figure 6). Hence, two distinct tasks can be recognised in the process of verification pattern refinement.

(i) *Reformatting* of verification pattern data, to fit the new format required at the next refinement level.
(ii) *Enrichment* of the same data, with the refinement information (see Figure 6), which does not appear in the

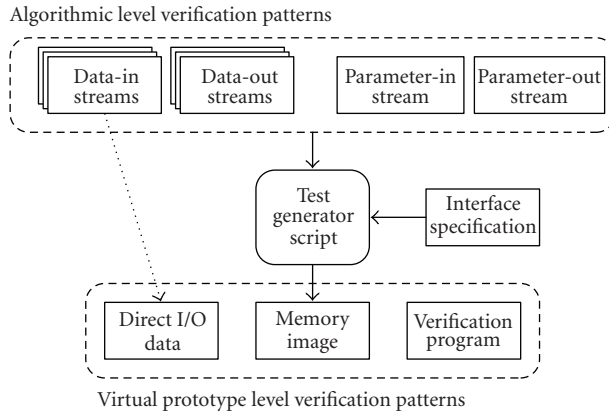

Figure 7: Structure of the environment for automatic generation of verification patterns.

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.

The environment for automated generation of virtual prototypes from algorithmic-level models presented in Section 2 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, performing both reformatting and enrichment of the verification patterns automatically.

3.1. Verification at algorithmic level

At the algorithmic level, the model of the system contains no architectural information and the partitioning of the system is 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 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 7.

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.

3.2. Verification at virtual prototype level

Use of a virtual prototype implies a highly heterogeneous system. 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 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. Hence, as the development of the system progresses, the VP becomes increasingly heterogeneous.

In this work, we focus on verification of system components assigned to hardware implementation, since they will be implemented as part of an HA block (see Figure 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:

(i) device under verification (DUV),
(ii) verification patterns,
(iii) verification program (runs on the DSP, applies the verification patterns to the DUV).

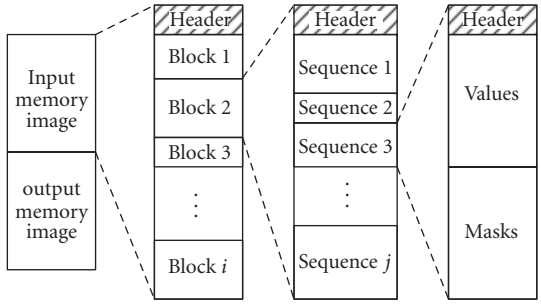

Figure 8: Structure of the memory image.

It is important to note that the structure of the hardware platform (see Figure 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 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.

3.3. Environment for automatic generation of verification patterns

The environment for automated verification pattern refinement presented here generates virtual prototype-level verification patterns from algorithmic-level verification patterns, as shown in Figure 7.

3.3.1. COSSAP verification patterns

The environment for algorithmic-level modelling considered in this work is COSSAP from Synopsys. Hence, the algorithmic-level verification patterns used also come from the COSSAP environment. As seen in Figure 7, they are divided into four sets of streams parameter in and out, and data in and out streams.

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 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, that is, is driven purely by the availability of input parameters and data, no further timing information needs to be contained in the streams.

3.3.2. 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 (see Section 3.3.3 for more details).

Functionality of the verification program is hence not dependent on the particular VP being verified. 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 used in the system and being employed to run the verification of any DUV.

3.3.3. 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 organised as shown in Figure 8. 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)

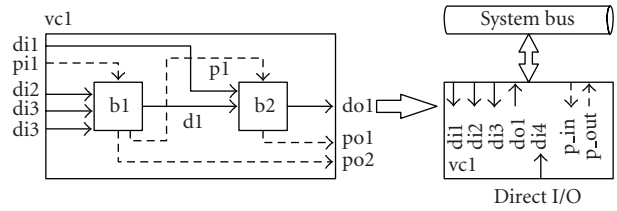

Figure 9: Model refinement of the virtual component vc1, from algorithmic level (left) to virtual prototype level (right).

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 "do not 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.

3.3.4. Direct I/O data

As already mentioned in Section 3.2, 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 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.

3.3.5. Interface specification

The interface specification (see Figure 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 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, 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.

3.3.6. 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 7. Its main function is to create the VP level verification patterns, that is, perform both steps in the verification pattern refinement process automatically (see Section 3).

In order to achieve this, the TGS creates the structure of the memory image as described in Section 3.3.3. The reformatting step of the verification pattern refinement process is achieved by interleaving the block-based structure of

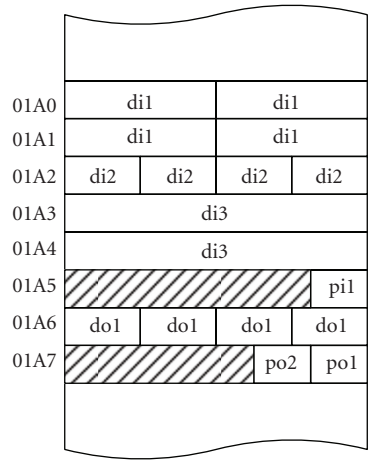

Figure 10: Register mapping of each data and parameter port of vc1.

```
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 11: Interface specification for the virtual component vc1.

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, that is, 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.

di1	di2	di3	do1	para_in	para_out
New block	New block	New block	New block	New block	New block
CD9A	1B	000B0855	22	pi1A	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 12: The COSSAP verification patterns for each port of vc1.

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. 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 9. Here the model of vc1 in the algorithmic modelling environment, such as COSSAP, is shown on the left. The component is made up of two subblocks, 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 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, as shown in Figure 6 as "Refinement Information." This architectural information includes the architectural 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 10. Hence, the bus interface between the component vc1 and the processor core 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 nonoverlapping (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 11. Here, it is specified that the input

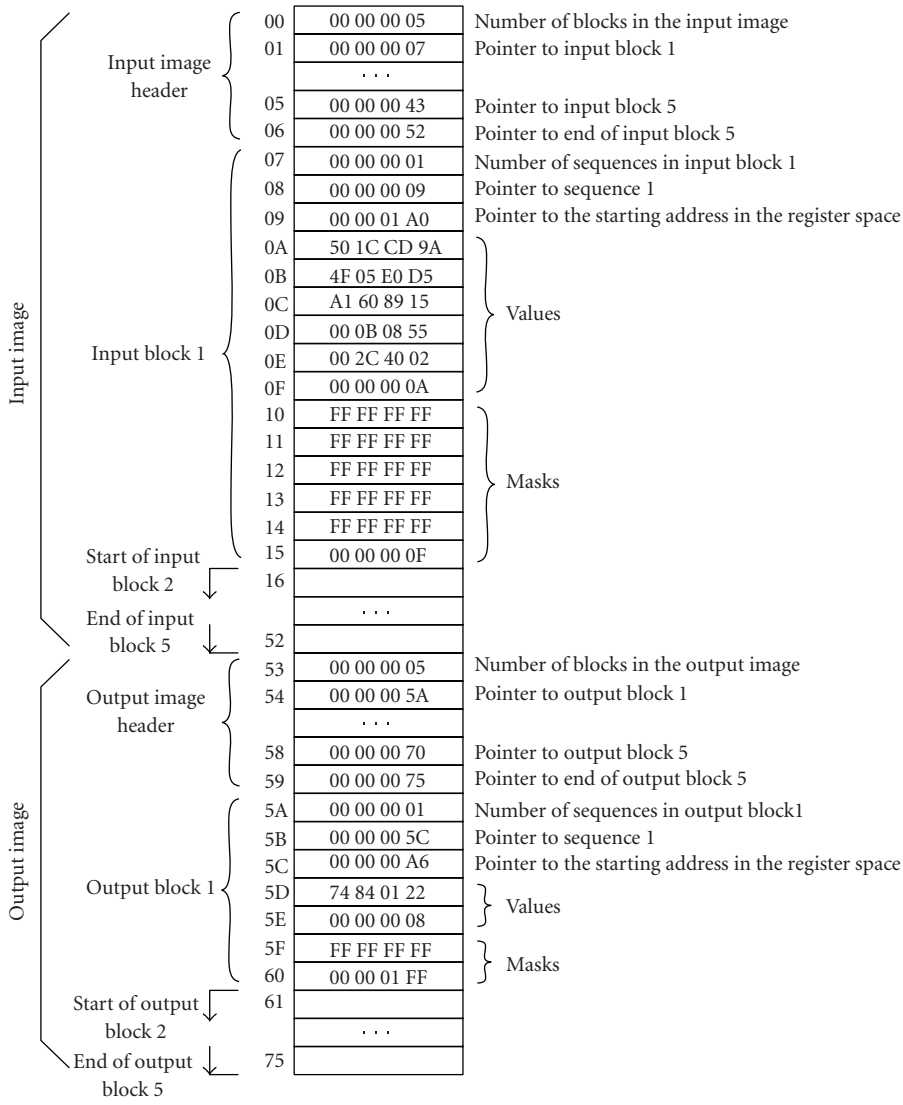

Figure 13: The structure and content of the memory image for the virtual component vc1.

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 are shown in Figure 12. As described earlier, each data input and data output port in 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 devided 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 13.

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 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.

4. CONCLUSIONS

In the rapidly changing and highly competitive field of wireless communication systems, minimizing time to market is a key requirement for any commercially viable product development. While virtual prototyping has proved to be one of the most effective techniques for achieving the required time savings, it is only with full automation that the maximal gains can be achieved.

The presented environment for automated development of virtual prototypes not only offers these maximal time gains, but also supports the virtual prototyping process comprehensively, in both the design and verification flows. In other words, the transition from the algorithmic-level to the corresponding virtual prototype is covered seamlessly by the presented environment, for both the model itself, as well as for the associated verification patterns.

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 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.

Furthermore, the verification strategy presented here has been implemented only for systems built around the StarCore DSP [20]. However, the modular nature of the verification environment ensures the applicability of the environment to systems build both around other processor cores as well as multiprocessor systems, with only minimal modifications and/or extensions. 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.

ACKNOWLEDGMENTS

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

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, 1965.

[2] R. Subramanian, "Shannon vs Moore: driving the evolution of signal processing platforms in wireless communications," in *Proc. IEEE Workshop on Signal Processing Systems (SIPS '02)*, pp. 2–2, San Diego, Calif, USA, October 2002.

[3] International SEMATECH, The International Technology Roadmap for Semiconductors, Austin, Tex, USA, 1999.

[4] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. IEEE*, vol. 91, no. 1, pp. 145–164, 2003.

[5] P. Belanović, M. Holzer, D. Mičušík, and M. Rupp, "Design methodology of signal processing algorithms in wireless systems," in *Proc. International Conference on Computer, Communication and Control Technologies (CCCT '03)*, pp. 288–291, Orlando, Fla, USA, July–August 2003.

[6] A. Hemani, A. K. Deb, 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*, vol. 5, no. 3-4, pp. 295–311, 2000.

[7] C. A. Valderrama, A. Changuel, and A. A. Jerraya, "Virtual prototyping for modular and flexible hardware-software systems," *Design Automation for Embedded Systems*, vol. 2, no. 3-4, pp. 267–282, 1997.

[8] N. S. Voros, L. Sánchez, A. Alonso, A. N. Birbas, M. Birbas, and A. Jerraya, "Hardware-software co-design of complex embedded systems: an approach using efficient process models, multiple formalism specification and validation via co-simulation," *Design Automation for Embedded Systems*, vol. 8, no. 1, pp. 5–49, 2003.

[9] R. Ernst, "Codesign of embedded systems: status and trends," *IEEE Des. Test. Comput.*, vol. 15, no. 2, pp. 45–54, 1998.

[10] P. Varma and S. Bhatia, "A structured test re-use methodology for core-based system chips," in *Proc. IEEE International Test Conference (ITC '98)*, pp. 294–302, Washington, DC, USA, October 1998.

[11] B. Stöhr, M. Simmons, and J. Geishauser, "FlexBench: reuse of verification IP to increase productivity," in *Proc. Design, Automation and Test in Europe Conference and Exposition (DATE '02)*, pp. 1131–1131, Paris, France, March 2002.

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

[13] P. Belanović, M. Holzer, B. Knerr, M. Rupp, and G. Sauzon, "Automatic generation of virtual prototypes," in *Proc. 15th International Workshop on Rapid System Prototyping (RSP '04)*, pp. 114–118, Geneva, Switzerland, June 2004.

[14] P. Belanović, B. Knerr, M. Holzer, G. Sauzon, and M. Rupp, "A consistent design methodology for wireless embedded systems," *EURASIP Journal on Applied Signal Processing*, Special issue on DSP enabled radio, 2005
[15] B. Knerr, M. Holzer, and M. Rupp, "HW/SW partitioning using high level metrics," in *Proc. International Conference on Computer, Communication and Control Technologies (CCCT '04)*, Austin, Tex, USA, August 2004.
[16] U. Bortfeld and C. Mielenz, "White paper C++ System Simulation Interfaces," Infineon, Munich, Germany, July 2000.
[17] The Open SystemC Initiative (OSCI), San Jose, Calif, USA, www.systemc.org.
[18] CoWare Incorporation, "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, Boston, Mass, USA, 2002.
[20] StarCore DSP, www.starcore-dsp.com.

P. Belanović received his Dr. tech. degree in 2006 from the Vienna University of Technology, Austria, where his research focused on design methodologies for embedded systems in wireless communications, virtual prototyping, and automated floating-point to fixed-point conversion. He received his M.S. and B.E. degrees from Northeastern University, Boston, and the University of Auckland, New Zealand, in 2002 and 2000, respectively. His research focused on the acceleration of image processing algorithms with reconfigurable platforms, both in remote sensing and biomedical domains, as well as custom-format floating-point arithmetic. Currently he is a Ph.D. candidate at the Vienna University of Technology, Austria, focusing on the design methodologies for embedded systems in wireless communications, virtual prototyping, and automated floating-point to fixed-point conversion.

B. Knerr studied communications engineering at the University of Saarland and the Technical University of Hamburg, Harburg, respectively. He finished the diploma thesis about OFDM communications systems and graduated with honours in 2002. He worked for one year as a Software Engineer for the UZR GmbH & Co KG, Hamburg, on image processing and 3D computer vision. In June 2003 he joined the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms at the Vienna Technical University as a Ph.D. candidate. His research interests are hw/sw partitioning, multicore task scheduling, static code analysis, and platform-based design.

M. Holzer received his Dipl. Ing. degree in electrical engineering from the Vienna University of Technology, Austria in 1999. During his diploma studies he worked on the hardware implementation of the LonTalk protocol for Motorola. From 1999 to 2001 he worked at Frequentis in the area of automated testing of TETRA systems and afterwards until 2002 at Infineon Technologies on ASIC design for UMTS mobiles. Since 2002 he has a research position at the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms at the Technical University of Vienna.

M. Rupp received his Dipl. Ing. degree in 1988 at the University of Saarbrücken, Germany and his Dr. Ing. degree in 1993 at the Technische Universität Darmstadt, Germany, where he worked with Eberhardt Hänsler on designing new algorithms for acoustical and electrical echo compensation. From November 1993 until July 1995 he had a postdoctoral position at the University of Santa Barbara, California with Sanjit Mitra where he worked with Ali H. Sayed on a robustness description of adaptive filters with impacts on neural networks and active noise control. From October 1995 until August 2001 he was a member of the Technical Staff in the Wireless Technology Research Department of Bell-Labs where he was working on various topics related to adaptive equalization and rapid implementation for IS-136, 802.11, and UMTS. He is presently a Full Professor for Digital Signal Processing in Mobile Communications at the Technical University of Vienna. He is an Associate Editor of the IEEE Transactions on Signal Processing and of the EURASIP Journal on Applied Signal Processing, and EURASIP Journal on Embedded Systems, and is elected as an AdCom Member of EURASIP. He authored and coauthored more than 180 papers and patents on adaptive filtering, wireless communications, and rapid prototyping, including 12 patents.