A Component Oriented Simulator for HW/SW Co-Designs

Alexander Paar
Institute for Program Structures and Data Organization (IPD)
Universität Karlsruhe
alexpaar@ieee.org

Haitao Du
Electrical Engineering & Computer Science (ECE)
University of California, Irvine
hdu@ece.uci.edu

Nader Bagherzadeh
Electrical Engineering & Computer Science (ECE)
University of California, Irvine
nader@ece.uci.edu

ABSTRACT
In order to extensively explore design space one has to specify a system on a very abstract level. Transforming a specification into a correct implementation is usually an error prone task. Moreover, one may want to specify a system on different abstraction levels. This work introduces a component oriented simulator approach that comprises AsmL based executable specifications. Test cases are automatically generated from the specification. Concurrent verification techniques are used to run the specification in parallel with the implementation. Feasibility of such rapid specifying techniques is proved by simulating a 3D graphics engine for the UC Irvine MorphoSys SIMD system-on-chip.

1. INTRODUCTION
Main systems engineering objectives are to design, develop, implement, and integrate a system as fast as possible and as correct as possible. In order to avoid wrong strategic decisions, one has to consider all possible solutions. This is why it is absolutely evident to extensively explore design space before any implementation starts. Time for design space exploration usually adds to the time that is needed for implementation and all succeeding steps. But time-to-market of a product is crucial considering consumer acceptance and financial margin issues. This paper introduces a rapid specifying approach that significantly eases the process of specifying a complex design. Rapid specifying techniques were used to modify the UC Irvine MorphoSys SIMD system-on-chip. A 3D graphics engine was developed in order to set up a true HW/SW co-design. Section 2 of this paper summarizes previous system specification approaches. Section 3 presents the MorphoSys architecture and a 3D graphics engine based on the MorphoSys system-on-chip. Section 4 gives an overview of the entire specification-emulator environment. Automatic test case generation is elucidated in section 5. Runtime verification use cases are presented in section 6. Section 7 concludes recent accomplishments.

2. PREVIOUS WORK
Recent approaches that address time-to-market and correctness issues are system specification languages that follow a specification driven design of systems. SpecC [1] is such a standardized specification description language. SpecC source code is used to specify and simulate a design on system level. SpecC compilers/synthesizers generate so-called SpecC handout code that is automatically converted into software source code (e.g., Java, C++) or hardware description languages (e.g., VHDL, Verilog). Assuming correct compiler and converter software, a correct system specification is transformed into a correct implementation. This approach may work well in theory. However, the generated implementation may not be much optimized and it may be inappropriate to specify the complete system on one abstraction level and in one and the same language. Hence disregarding that one may want to specify a system on several levels of abstraction. Moreover, SpecC itself does not have a big community yet and many experienced hardware and software engineers may prefere languages and development environments to which they are already accustomed. Approaches to solve long simulation times are rapid prototyping systems (i.e., hardware in the loop). A hardware design is synthesized to a reconfigurable logic device (i.e., FPGA, CPLD). This can speedup simulation time tremendously but the results are mostly inaccurate and technology dependent due to typical tradeoffs of fine grain reconfigurable logic devices such as different timing delays for bus wires and completely different routing results with every new synthesis cycle. A third approach is software-based simulation of processor architectures. Such simulators accept binary or hex input that is generated by target assemblers/compilers for the specific platform and allow clock cycle, register accurate simulation. A major tradeoff is the very structural description of the target platform. To change the behavior of such a simulator, it is mostly necessary to model a complete new hardware structure. To put it in a nutshell, every existing specification/simulation tool has its specific advantages but also major tradeoffs.

3. A SIMD 3D-GRAPHICS ENGINE
3.1 MorphoSys SIMD System-On-Chip
The UC Irvine MorphoSys architecture [2] is a coarse grain, integrated reconfigurable system-on-chip that follows the SIMD computation model. It incorporates a reconfigurable array of processing elements, a RISC processor core, a memory interface unit, and an 8x8 array of SIMD processing elements. MorphoSys processing elements are in the following referred to as Reconfigurable Cells (RC).

A Reconfigurable Cell comprises a 32-bit context register that contains the SIMD instruction (Context Word) for the current cycle, two input data multiplexers, and a combinational network that includes an ALU, a multiplier and a shifter. Data is stored in a set of RC registers and in an output register for inter cell data exchange. Data width within one RC is 16 bit. The Context Words for the RC Array are stored in Context Memory. The controlling component of MorphoSys is a 32-bit processor, called TinyRISC, based on the design of a RISC processor in [3]. TinyRISC handles general-purpose operations and controls operation of the RC Array through special instructions added to its ISA. It also initiates all data transfers to or from the Frame Buffer and configuration program load for the Context Memory.
MorphoSys is a reconfigurable SIMD architecture. Reconfiguration is the process of reloading configuration programs (context). To provide depth of programmability, Context Memory stores different contexts for each RC. Thus, each processing element can be considered as a reconfigurable processing unit having several contexts concurrently residing in the system. Contexts are broadcasted row wise, column wise or to the entire array. The first level of the RC Array interconnection network is the nearest neighbor layer that connects the Reconfigurable Cells in a 2-D mesh. The second layer of connectivity is at the quadrant level (a quadrant is a 4x4 RC group), which provides complete row- and column connectivity within a quadrant. Therefore, each RC can access data from any other RC in the same row/column within the quadrant. At the highest or global level, there are buses that support inter-quadrant connectivity. These buses are also called express lanes and they run across rows as well as columns. These lanes can supply data from any one Reconfigurable Cell (out of four) in a row (or column) of a quadrant to other Reconfigurable Cells in same row (or column) of the adjacent quadrant. This results in enhanced performance for application kernels that involve high data movement.

By incorporating coarse grain, dynamically reconfigurable cells and supporting depth of programmability, MorphoSys had already offered a level of programmability that was sufficient for a wide range of regular data-parallel applications [4]. Using a component oriented simulator, a novel predication scheme [5, 6] was developed in order to implement two main categories of execution autonomy: operation autonomy and communication autonomy.

3.2 A 3D Graphics Engine

A 3D graphics engine that renders boundary-represented triangles was implemented in order to evaluate benefits and efficiency of a predicated SIMD architecture as introduced in the previous subsection. Fig. 3 depicts the operations that are performed to display a three dimensional world on a two dimensional screen.

MorphoSys RCs do only provide integer arithmetic ALUs. However, pure integer arithmetic is insufficient for an implementation of a rendering engine. We developed a 32-bit S.15.S.15 fixed-point number format that comprises two distinct 16-bit words to fit MorphoSys ALU capabilities. Both 16 bits were used for the integer- and the fractional part. Section 4 describes how a Microsoft Direct3D based client application was used to verify that images, which had been rendered using fixed point arithmetic, were visually pleasing, too.

4. SIMULATION ENVIRONMENT

4.1 Microsoft COM

A main objective of the developed simulator was to support easy incorporation of both application specific components as well as commercial off-the-shelf software. Microsoft COM [7] is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM allows an object to expose its functionality to other components and to host applications. It defines both how the object exposes itself and how this exposure works across processes and across networks. COM components can have been written in different languages, and may be structurally quite dissimilar. We decided to use Microsoft COM for component integration since it is a de-facto industry standard that is supported by a wide range of existing applications.

4.2 Mathworks Matlab

Mathworks Matlab [8] is a software for technical computing. It provides a console-based interface that can be used to define variables and to perform numerical operations. It is a convenient tool that hides data type and memory allocation issues from the user and is thus very applicable to very easily specify an
algorithm in a rapid specifying manner. Matlab on Microsoft Windows supports COM automation server capabilities. Thus, Matlab can be launched and controlled by any program that can act as a COM automation controller. Every command that can be entered using the command prompt can also be issued via the Matlab COM interface. In this work Matlab was used to generate graphics transformation matrices.

4.3 Mulate MorphoSys Simulator
Previous work on the MorphoSys architecture included a complete programming environment for the MorphoSys M1 implementation. The main components of this tool set are depicted in Fig. 4.

4.4 Microsoft AsmL
AsmL is the Abstract State Machine Language [11, 12]. It is an executable specification language based on the theory of Abstract State Machines [13]. Abstract State Machines are a mathematically precise model to describe the dynamic behavior of complex systems (i.e. their evolving runtime state). The ASM computational model is similar to the signal assignment rules in VHDL. Within one state (simulation cycle) several variables (signals) are assigned values synchronously. All updates (assignments) become visible in the next state (next simulation cycle).

Fig. 6 and 7 show a short ASML-, VHDL listing, respectively. The bold statements in the VHDL listing will assign signal y1 and y2 the same value only if the input signal x1 has the same value for two consecutive rising clock edges since the signal assignment y1 <= x1 is not visible for y2 <= y1 until the next clock cycle. The same holds true for the ASML code. The variable y2 is not
assigned the value of $x1$. Instead it will always be zero. The ASML keywords `machine` and `step` define an Abstract State Machine and when its variable updates are to occur, respectively. AsmL is embedded into Microsoft Visual Studio .NET. It uses XML for literate specifications. The AsmL compiler generates C++ source code and bindings for native COM and automation interfaces. As for any other COM software, the Microsoft MIDL compiler is used to generate a type library that can be used to create wrapping classes in Visual C++ projects. AsmL source code can be executed via compiling and linking it to an executable file (.exe-file) or a dynamic link library (.dll-file). This is why a specification written in AsmL is called executable specification. Executable specifications have a single, unambiguous meaning. They are minimal in that sense that although they define everything that is part of the chosen level of detail, they leave unnamed everything that is outside that level of detail. ASML specifications are limited to the constraints and the behavior that all correct implementations of a system have in common. This is a major difference to traditional programming languages.

An executable specification is useful both before as well as after a system has been implemented. By exploring design space before implementation one can answer questions like: Does my design do everything it is intended to? How do all features interact? Are there any unintended behaviors? Once a system has been implemented one can run the specification in parallel with the implementation to find out whether the implementation produces the expected results. Furthermore, AsmL can automatically generate behavioral test cases from the specification. The following section elucidates how AsmL was used to model MorphoSys RCs, arrays of RCs, and sequences of SIMD instructions.

### 4.5 AsmL Component Specifications

AsmL provides many features that make it possible to specify a system as general as possible. This is why the essential component of the simulation environment, the predicted MorphoSys RC and its assembly in an SIMD array, were modeled in AsmL. However, much of this generality would have been obliterated if AsmL had been used to model the structure of particular implementation related hardware blocks. Traditional simulators follow this approach to represent every hardware basic block by a class of an object-oriented language and to aggregate this set to assemble the overall architecture.

There are two major tradeoffs of this approach. Firstly, the specification does not only model essential elements that define the actual computer architecture (i.e. the hardware/software interface) but also implementation related units that are neither accessible nor visible to a software programmer. Secondly, changing the structure of one basic block may affect the behavior of the entire instruction set. This is why the chosen AsmL based specification does not describe the complete structural model but only those system elements that are visible to the programmer. The system itself is not specified structurally per basic block but rather per instruction.

#### 4.5.1 MorphoSys Reconfigurable Cells

Every RC instruction is represented by an Abstract State Machine that operates on global variables given in Fig. 8. These variables define external data input, visible data registers, and ALU flag generation- and predication related attributes. Not all of these variables have their equivalent counterpart in hardware but are essential for a complete specification. An example for such a “virtual” attribute is the `clockCycles` variable.

```plaintext
// Clock cycle
var clockCycles as Integer
// External input
var exI as Short
var exJ as Short
var exL as Short
var exM as Short
var exR as Short
var exT as Short
var exC as Short
var exB as Short
var exP as Short
var exU as Short
var exD as Short
var exH as Short
var exV as Short
// Output register
var rcOut as Short
// Express lanes
var exHE as String
var exVE as String
// Register set
var r0 as Short
var r1 as Short
var r2 as Short
var r3 as Short
var r4 as Short
var r5 as Short
// 32-bit register
var reg32 as Integer
var grdPred as Boolean
var qPred as Boolean
var evalEqs as String
var strPredD as String
var strPredB as String
var p7 as Boolean
var p5 as Boolean
var p4 as Boolean
var p3 as Boolean
var p2 as Boolean
var p1 as Boolean
var p0 as Boolean
var evalEqs as String
var labelEq as String
var labelPred as Boolean
var strPred as String
var strPredD as String
var strPredB as String
var p7 as Boolean
var p5 as Boolean
var p4 as Boolean
var p3 as Boolean
var p2 as Boolean
var p1 as Boolean
var p0 as Boolean
var evalEqs as String
var labelEq as String
var labelPred as Boolean
var strPred as String
var strPredD as String
var strPredB as String
var p7 as Boolean
var p5 as Boolean
var p4 as Boolean
var p3 as Boolean
var p2 as Boolean
var p1 as Boolean
var p0 as Boolean

Figure 8. RC Attributes.
```

Fig. 9 shows the AsmL source code for the read predicated ADD instruction. The top-level hierarchy of the MorphoSys RC behavior as well as every instruction is modeled as an Abstract State Machine. Thus, every instruction defines sub states within the MorphoSys RC Abstract State Machine. According to the ASM computation model, each call of an instruction function changes the interpretation of subsequent instruction functions. The given example of such a behavior oriented approach makes clear that it is very easy to modify the specification of an existing opcode or to add the behavior of new instructions that may even employ further computer architecture attributes. One simply has to model a behavior representing Abstract State Machine for that new instruction and to add required global attributes.

Part of the MorphoSys RC component is an AsmL class that represents an Abstract State Machine to accept a defined SIMD assembly language instruction set. MorphoSys Context Words were given in Backus-Naur forms. Abstract State Machines define automats $M$ that accept the BNF defined Context Words $w$ el of $L_{cw}$ that present the input language $L_{cw}$. This ASM was also extended with assembler capabilities in order to produce an output language $L_{out}$ with an output word $w$ el of $L_{out}$ (binary or hex values) for each legal input Context Word. Thus, AsmL again proved to be an efficient way to define languages as well as to parse, accept, and translate them. AsmL COM integration was used to make the RC component available to the MorphoSys RC Array component described in the next paragraph.
4.5.2 MorphoSys RC Array

This component assembles an 8x8 MorphoSys RC Array by instantiating the MorphoSys RC COM component. The array component simply provides functions to set data input for each RC instance and to access output data from them. It further exports Context Word broadcast functions for row-, column-, and array broadcast. The specification defines control flow constraints, namely that a Context Word is to be broadcasted row wise, column wise, or array wise. It further specifies how RCs are connected within the RC Array. The specification does not contain data flow constraints such as bandwidth limitations between RC Array and FrameBuffer. Definition of those parameters was left to the utilizing application since only the application layer contains enough information to decide what bandwidth and array access flexibility is needed (e.g. in what manner data has to be transferred). Thus, it was possible to assume any possible FrameBuffer capabilities.

4.5.3 Context Word Sequences

AsmL was used to specify both the MorphoSys SIMD hardware as well as the executed Context Word sequences. Software specifications can be partitioned into control- and data flow graphs. Control flow refers to what Context Words are broadcasted to the SIMD array in what manner. Data flow refers to the input data on which operations are performed. Fig. 10 shows an excerpt from a Context Word listing.

Each simulated clock cycle comprises of a data update phase and a Context Word execution phase. Firstly, all necessary input variables are assigned their values. The second step keyword in the listing above makes these updates visible to the following Context Word broadcast. How the Array Control Unit later on implements data flow instructions is not part of the specification. Instead, the RC Array component provides two functions to arbitrarily set input data for all FrameBuffer connected RC inputs.

4.6 A Direct3D Client Application

The MorphoSys 3D graphics engine does employ fixed point arithmetic instead of floating point number formats that would provide higher precision. This is why system validation had to include not only logical correctness but also quantitative rounding issues. In contrast to logical correctness, it is not appropriate to classify quality of an image as correct or incorrect. As an alternative, a human observer has to evaluate whether generated images are “visually pleasing”. A Microsoft Direct3D [14] based client application was developed to display contents of Microsoft X-files. Such X-files, which contain graphics data (i.e. boundary represented triangles), were processed using the Direct3D graphics engine as well as component oriented simulator components as introduced above. Direct3D functionality was successively replaced by simulator components. Thus, one could easily compare differences between “perfect” Direct3D generated images and image output of the MorphoSys 3D graphics engine. All in all, quality proved to be absolutely satisfactory for screen resolutions up to 512x512 pixels. Fig. 11 shows a screen shot of the Direct3D client application.
5. TEST CASE GENERATION

Testing hardware components is a tedious and time consuming task. Simulation runs may take a long time to complete. This is why hardware-in-the-loop test environments try to speed up execution by employing hardware components to emulate critical system parts. Our work presents a complementary approach. Beyond speeding up simulation time itself, we try to decrease the number of simulation runs that are necessary to meet certain quality requirements. AsmL was used to model MorphoSys hardware units and sequences of SIMD instructions. The latest release of AsmL for Microsoft .NET [15] contains a tool for automatic test case generation. It can be used to automatically generate test cases from an AsmL model using various algorithms, and to use such test cases to perform a conformance test against an actual implementation. Firstly, one has to find interesting sequences of method calls. Secondly, interesting parameters for each method call have to be found. Finally, a conformance test is performed against an implementation as described in section 6.

When testing the SIMD instruction set of a stand-alone RC (i.e. one that does not interact with interconnected RCs in an RC Array) each test sequence consists of a sequence of Context Words and external input data that are issued to the RC. The AsmL test tool generates a finite state machine (FSM) that is then traversed using a Chinese Postman Tour. Automatic parameter generation was used to test basic hardware units such as shifter or parity creation. Fig. 12 shows the AsmL test tool after a FSM build. The left most part depicts a textual representation of all states. The rightmost part displays a graphical view that utilizes the optional graph visualization component Graphviz [16] from AT&T research labs.

Current work includes augmenting the entire RC Array model with sufficient auxiliary information to facilitate automatic test case generation for the overall system.

6. RUNTIME VERIFICATION

6.1 Runtime Verification Technique

Employing the introduced component oriented simulator approach, the complete system was assembled of a set of interface specifications. Any component implementation is a behavioral refinement of its interface specification. This section proposes the use of AsmL based executable specifications and a runtime monitor to check for behavioral equivalence between a VHDL based implementation of a component and its specification. Thus, addressing the problem of ensuring that a particular component does indeed implement its specification. As introduced in section 4, all simulator components export their functionality via Microsoft COM interfaces. A client application is not at all aware whether it calls a specification- or an implementation component. This is why runtime verification introduces a proxy component as depicted below. A model component spies on the results that are generated by the implementation (see [17, 18] for a general introduction to runtime verification and conformance checking).

Because access to a COM component is initiated by the operating system it is possible to intercept a client’s call to a server component. This call is then forked by the proxy to both the server- as well as the model component. As long as the proxy component returns a valid interface it is not possible for the client component to distinguish whether the handle is to the actual implementation, S, or to the specification model, M. Runtime verification means that from the client’s point of view, the observed behavior of the model is indistinguishable from that of
the server, i.e., they are behaviorally equivalent. Because this is a dynamic check, it means they are equivalent only on the observed behavior; ideally the specification allows more behaviors. An implementation restricts its behavior, for reasons of efficiency.

6.2 Runtime Verification Components

The behavior of a predicated Reconfigurable Cell was specified using AsmL and implemented as a set of VHDL entities. A runtime verification environment comprises AsmL specifications (model component), VHDL implementations (server component), and a client application that communicates via a proxy.

Fig. 14 shows a screenshot of the client application. The upper portion of the GUI presents the executed Context Word sequence. The middle part displays computer architecture RC attributes and bottommost there are buttons for user interaction. Firstly, one has to instantiate the RC AsmL COM component. The next step is to load a test suite Context Word sequence.

Fig. 15 depicts the input format that specifies not only the Context Word that is to be executed but also external RC input from the Framebuffer and inter cell bus connections.

Since the VHDL design software [19, 20] that was used for VHDL simulation does not support COM automation, file I/O was employed for data exchange. Two simulation runs are needed to perform runtime verification for one certain test suite. Due to performance reasons, proxy functionality was integrated into the client application and was not implemented as a dedicated component. The first run automatically generates a VHDL test bench stimuli file from the input Context Word sequence. For that purpose, the AsmL RC specification was extended with assembler capabilities that translate parsed Context Words into binary sequences. These strings are incorporated into the automatically generated stimuli file as exemplified in Fig. 16.

Fig. 17 shows a list file that displays signal values of all five RC output registers at 5 relevant simulation times. The second verification run reads this list file and after each computational step compares the AsmL- and VHDL based output register values. This approach eased simulation of the VHDL implementation tremendously. This holds especially true for very long Context Word sequences that were to be verified. One does not have to check waveforms and signal values at each clock cycle manually but one could instead run the entire test suite automatically. Simulation time of each verification error is displayed at the end of the verification run. One simply has to check waveforms at these times.

Runtime verification was done in a totally automated manner since simulated test cases were automatically generated by the Microsoft AsmL test case generation tool as elucidated in section 5. Stimuli files were inferred from a minimal set of AsmL specification derived test cases. This approach reduced testing errors and thus led to a high efficiency and a significantly improved correctness of design validation.

7. CONCLUSIONS & OUTLOOK

This paper introduced rapid specifying by means of a component oriented simulator. A complex HW/SW co-design was modeled using a variety of development tools that were all glued together via Microsoft COM as a common binary standard. Distinct parts of the overall system were specified on very different abstraction levels in order to set up a complete simulation environment as soon as possible. Hence, it was feasible to test both the overall system as well as each single component not only with artificial test suites but with real life application data. Right from the beginning of the system development process, one could view the fully processed image.
Microsoft AsmL was used to incorporate executable specifications of crucial system parts that were subject to extensive design space exploration. Simulator components were successively replaced by their respective implementations. Conventional programming- and hardware description languages were used to develop implementations according to higher level specifications.

Conformance testing was eased significantly by utilizing the AsmL test tool to automatically generate test cases from a given specification model. VHDL test bench input was automatically generated from real live application data. Using runtime verification techniques, AsmL based specifications were executed in parallel with VHDL simulator runs. A graphical testing application highlighted VHDL simulation times when mismatches occurred between specification and implementation. Thus, efficiency of simulating and testing was increased considerably since both generation of test suites as well as conformance testing took place automatically without any user interaction.

The presented rapid specifying approach proved to be feasible and efficient, assuring quality and correctness of the final product.

8. REFERENCES


