2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

A Model-Based Reusable Framework to Parallelize Hardware and Software Development

Jouni Sillanpää, Nokia, Oulu, Finland (jouni.sillanpaa@nokia.com)

Håkan Pettersson, MathWorks, Stockholm, Sweden (hpetters@mathworks.com)

Tom Richter, MathWorks, Munich, Germany (trichte@mathworks.com)

***Abstract*— The traditional ASIC/SoC development process uses the waterfall methodology, where verification occurs at the end of the design process and comes with some significant difficulties. It leads to long verification closure loops, where bugs introduced early in the process are found at a very late stage, leading to higher costs and possible ASICs or SoCs manufacturing delays. Today's modern SoC development workflows address this issue by starting verification activities before the implementation phase [1][2], often referred to as "verification shift-left." Several discussions about "software shift-left": how to enable software design (and, consequently, hardware/software co-verification) have arisen recently in the pre-silicon design phases using techniques that allow "virtualization" of the hardware being designed [3][4]. This paper outlines a model-based framework where both hardware and software are modeled in the same environment, and the hardware model is used as a virtual platform for software development, allowing concurrent development. This framework allows software development to occur significantly earlier than more traditional workflows and enables better collaboration between teams. Both hardware and software bugs can be found early, and using code generation techniques enables automatically generated C/C++ code from the software models directly to pre-silicon verification. This leads to a significantly shortened time-to-market and better overall system quality.**

Keywords—Model-based design, Verification, Code Generation, Virtual Prototypes; Hardware/Software Co-design

I. INTRODUCTION

The described approach has been applied to the Nokia 5G Digital Front End (DFE) design, which conditions the base station radio unit SoCs carrier signals before being sent to DACs, the power amplifier, and the antenna. A typical DFE characteristic is its algorithmic complexity, consisting of digital up and down conversion, digital pre-distortion, crest factor reduction, channel filtering, gain control, composite carrier combination and separation, and many others. Most of these operations are implemented as complex signal processing algorithms running at high sample rates. An additional DFE characteristic is the need for configurability. In fact, designs may need to support several wireless standards and different configurations for network operator needs. These DFE characteristics lead to a very high implementation complexity. For instance, the design area can equal several hundreds of millions of gates, and a full register configuration consists of several megabytes. This complexity is often accompanied by hard time-to-market pressure and competitive solution demand. These 5G SoC verification challenges were also addressed in a DVCON Europe 2022 keynote [5].

The SoC development process also introduces hard pre-silicon verification deadlines because of the long lead times for getting the actual physical chip samples after the tape-out. Using test scripts/code is the fallback solution if software is unavailable, which is wasted labor and opportunity for shift-left testing. "Shift-left" refers to moving quality activities, such as hardware verification, software testing, deployment, releasing, etc., earlier in the project timeline. It is based on the observation made famous by Barry Boehm [6] that the cost of fixing a defect rises exponentially the longer it takes time to find that defect. The traditional waterfall process would be inefficient under these conditions, given the cost it imposes on design iterations in terms of time and effort.

The authors defined a model-based workflow to overcome these challenges. Hardware and software are modeled within the same environment, allowing development to be carried out simultaneously and started as early as possible. Coupled with C/C++ code generation techniques, this enables maximum reuse of other teams' contributions to the model and fast software deployment for hardware verification in pre-silicon phases. This ensures that effective hardware verification takes place before critical deadlines, using the same software in the

final product. This significantly increases software quality and yields cost and schedule benefits according to shift-left testing principles. This also increases the hardware verification coverage, as the more mature software allows more thorough pre-silicon environment testing – this simply cannot be done in models due to design size.

A stable interface is a key enabler for starting complex design software development with many parameters. Traditionally, this has been the register interface. However, a parameter-based abstract low-level API is used instead to begin before such details are available. This API is explained in more detail in Chapter III.

II. MODELING FRAMEWORK

Figure 1 depicts the Model-Based Design workflow in Nokia 5G development, with different modelling phases moving from high to low abstraction, finally targeting real hardware or pre-silicon environments like an emulator. Phases move forward as time progresses, but it is also possible to move back to lower abstraction to analyze problems found later in development. This is enabled because the use case configurations are common for all phases except the level of details in model changes. The medium abstraction phase is important from the software development view because it defines the functional split, parameters, and their types between hardware and software. This leads to defining the software algorithms and the hardware abstraction layer described more thoroughly in Chapter III. This enables software development teams to start their design process early in the MATLAB/Simulink model and run complex closed-loop simulations on key use case configurations before any hardware is available. Software can be deployed quickly with the help of code generation tools to pre-silicon environments, such as emulators or FPGA prototypes, when available. As the model acts as a golden reference for RTL verification, it is already verified that the software model provides correct configurations, and the pre-silicon activities can be more focused on system integration and longer test scenarios.

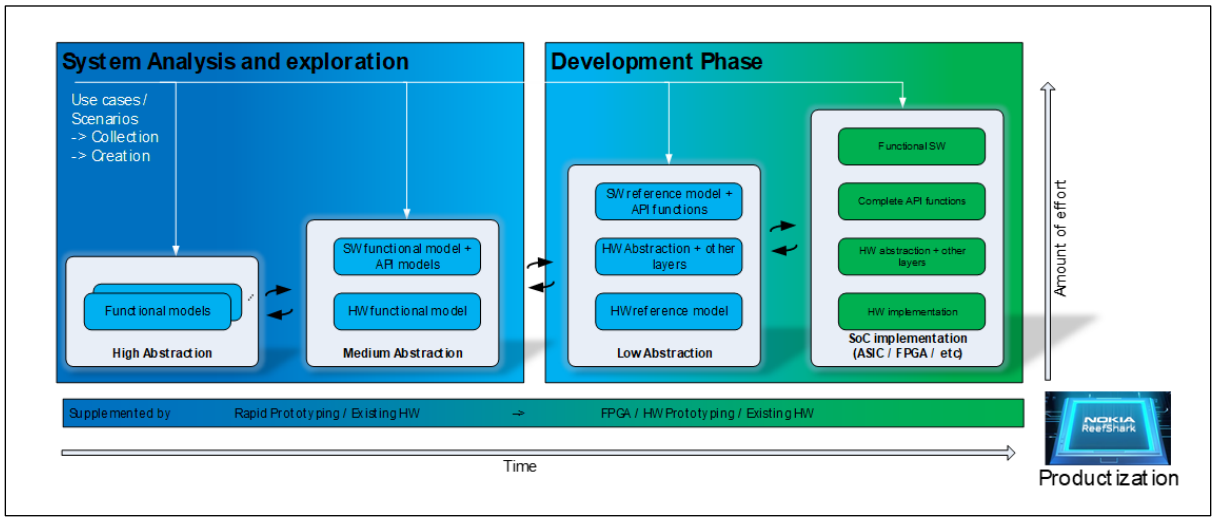

Figure 1. Model-Based Design workflow in Nokia 5G

III. ABSTRACTION LAYERS

One of the key elements in using virtual models before RTL is available is creating a Hardware Abstraction Layer. This enables software to use low-level configuration API and hide the actual register interface, which comes later in the RTL release. With thousands of configuration parameters, special care needs to be taken to avoid additional mapping and synchronization between the abstract parameters and registers. The principle behind defining abstraction layer parameters is that they should be defined with the same level of precision as the signal format used between hardware blocks. They can be used as the basis of register content if they are well-defined according to the algorithm's needs. Abstraction layer parameters can be considered the logical representation and

registers as the physical representation of the same content. Of course, hardware can have parameters the model does not have, but not vice versa. This way, a hardware extension API can be made in addition to the abstraction layer to handle the hardware-specific details, like enable-based signaling. Figure 2 shows the framework's software layer structure. The Abstraction Layer API is the interface that has different implementations based on user needs. The Abstraction Layer to Model Implementation can be automatically generated by the tool and used when running the model. The Abstraction Layer to Register Implementation, which maps the parameters to register values, can be used when running with hardware (i.e., generated code). This needs to be manually created into the template generated by the tool when the register definitions are available. Finally, an extension API can be created manually for configurations that only exist in hardware.

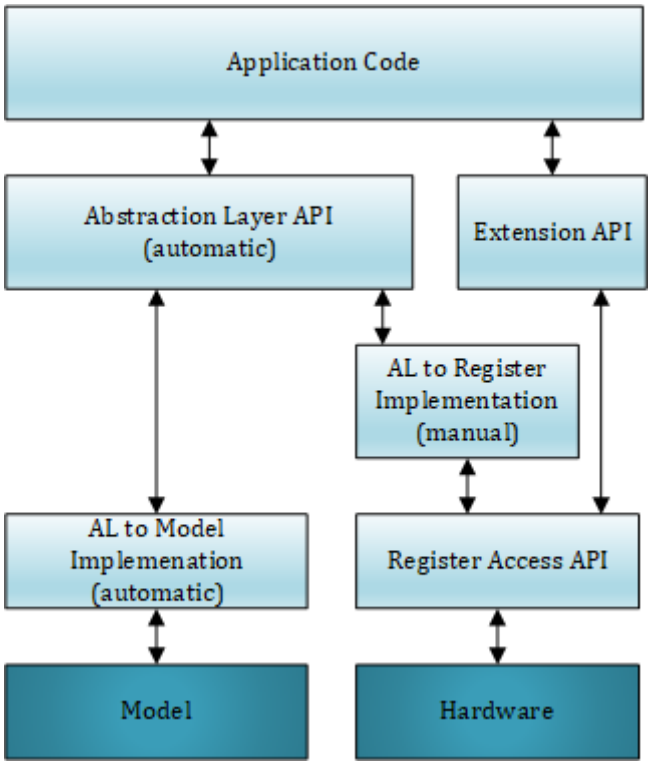

Figure 2. Structure of software layers in the design framework

A special custom tool was created in collaboration with MathWorks for the Abstraction Layer creation. The tool generates the output shown in Figure 3. It takes carefully defined, fixed-point type parameter inputs as MATLAB structures and transforms those into an Abstraction Layer API. Each parameter will get corresponding get- and set-functions with documented fixed-point types for each parameter. As a result, the software team gets an API for early software development based on one specification, the MATLAB structure data type. It can be used in the MATLAB/Simulink environment and real software with the help of code generation. Since this API hides any details underneath, it is future-proof for adapting this solution to other virtual platforms. It should be noted that this solution is not just about tooling but also about creating a single point of parameter specification together with modelling, hardware, and software teams. The better the specification, the fewer changes need to be made during the project. It may also potentially avoid struggling with different model, software, and hardware versions.

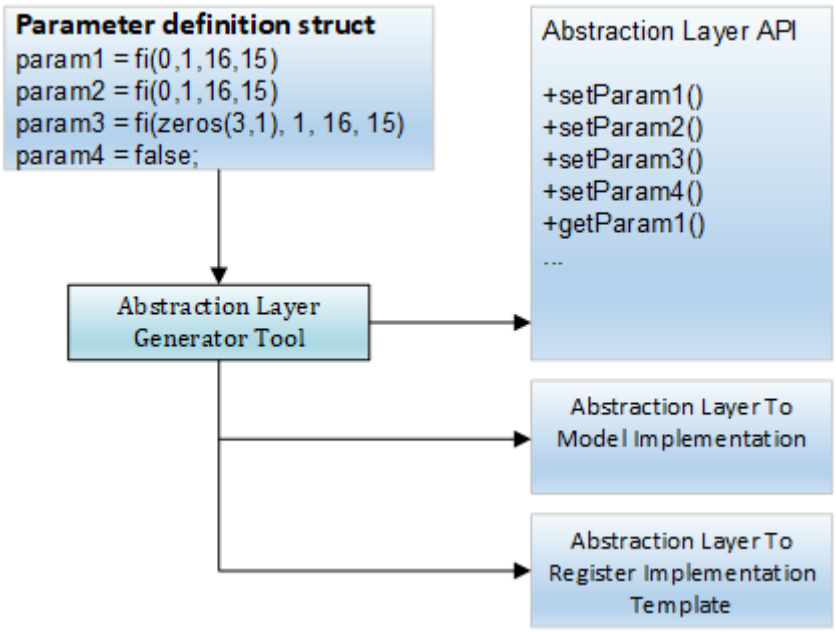

Figure 3. Hardware Abstraction Layer tool flow

IV. TOOLS USED FOR SUCCEEDING IN SHIFT-LEFT

The Model-Based Reusable Framework presented in this paper was developed in close collaboration between Nokia and MathWorks using tools from the MathWorks product portfolio. The high-abstraction functional models were programmed in MATLAB using an object-oriented design approach. Therefore, a code organization level could be achieved that helped with maintainability, reusability, and collaboration from the beginning.

The functional and reference software models could be derived from the original MATLAB code. System Objects – an object-oriented programming construct used for building, simulating, and implementing dynamic systems – were the base of the configuration-intensive part of the software reference model later generated into C++ code. MATLAB Coder was used to generate function-call-based C and C++ from MATLAB code and preserve the same kind of structure and interface of the MATLAB code in the generated code.

The control part of the software reference model was designed in Simulink to allow for closer hardware implementation, including discrete-time simulation. Integrated state machines for real-time control were modeled in Stateflow, which provides a graphical, powerful, and intuitive environment for modeling, simulating, and implementing complex systems with dynamic behavior. Also, the hardware reference design was modeled in Simulink and controlled through the MATLAB API and Stateflow charts, as seen in Figure 4. Fixed Point Designer helped define custom data types for signals within the design and for the parameters.

Embedded Coder is used to generate real-time, optimized, target-specific, and scheduling-based C code since the control part of the software is targeted to run on a real-time processor. The coder product also has many more features, like software-in-the-loop (SIL), used to verify the generated code independently of MATLAB/Simulink. Automatically generated reports and traceability between requirements, models, and generated code helped to review and document the output.

Furthermore, the verification is fully automated by using a class-based unit test framework that can be used for full simulations and testing the hardware abstraction layers. Continuous integration (CI) is one of the key practices in modern agile development processes. MathWorks supports this by providing an easy integration to CI tools like Jenkins. This enabled running simulations, tests, and even code generation continuously controlled by the CI tool and helped to detect issues early and to run lengthy operations, for example, overnight.

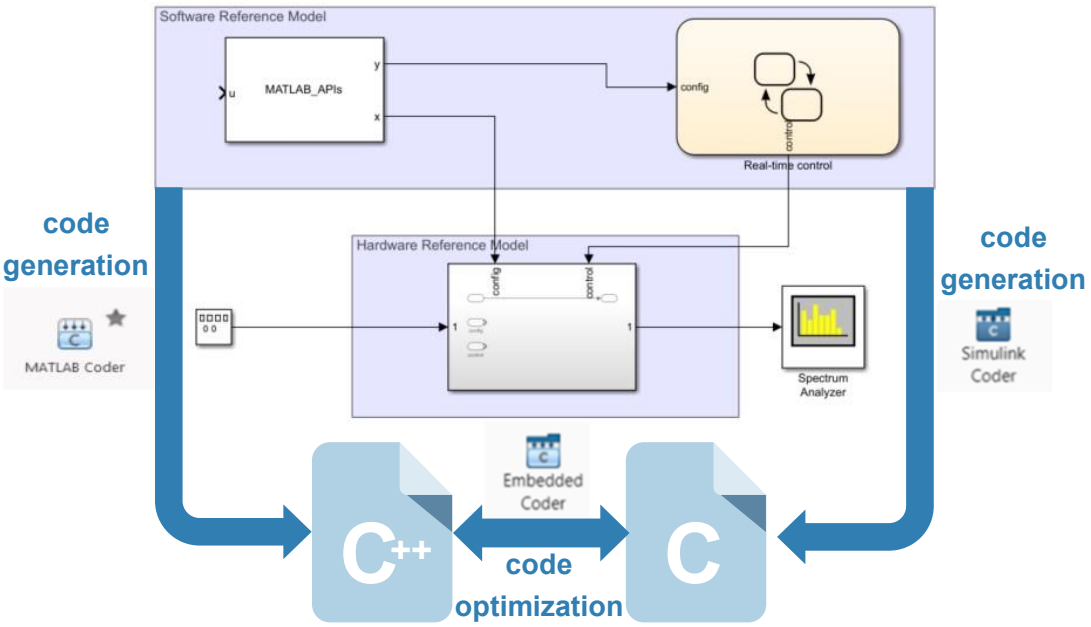

Figure 4. Simulink based design flow for the HW and SW reference model

V. RESULTS

We estimate the described model-based framework, coupled with code generation techniques, shortened the development cycle by about 30-40%. The projects are long, and there are variables that make direct comparison to previous projects difficult due to the DFE ASIC's complex nature. These variables include resourcing, requirements, organizational structure, and available equipment. However, the most important goal of executing verification using real production software was achieved for the first time with key test cases before the critical milestones. This has enabled more thorough testing with less wasted effort and increased software quality due to the shift-left testing. This quality improvement will yield accumulative benefits as building a base station product will continue with many additional software layers and customization for multiple product variants.

There were also other positive findings from the model-based approach. Because of the common model used for algorithm, hardware, and software development, there was improved collaboration and communication across the teams. The model enables teams to analyze problems found in pre-silicon testing and find more suitable parameters and new use cases. Also, the software shift-left testing starts already in the model. The same software model is used for hardware model configuration, testing it for algorithmic correctness, and providing test vectors and configurations for RTL verification. Therefore, the software model has already undergone many testing steps before entering the code generation phase.

Another key aspect is reusability. After the initial investment in model and framework building, these can be reused to build new products by scaling the design, adding or removing parts, and retargeting the code to other processor environments (i.e., switching from C++ to C). This makes new product development closer to branching than building everything from the ground up. There is also the reuse of work effort as algorithm designers can already implement changes in the parametrization of certain design parts, and the software team only needs to integrate the generated code.

VI. FUTURE ENHANCEMENTS

In the application presented in this paper, the design focus was on the algorithmic complexity and design configuration for a particular use case. Since the algorithms were already developed in MATLAB, it was natural to use the same design environment and reuse the existing design as a starting point for software development. However, having another virtual platform for integration and testing would be beneficial before going into pre-silicon environments, like emulators, FPGAs, or sample boards. These environments are tightly coupled with the hardware schedules and generally have limited resources that are expensive for trivial testing. A virtual platform could be a solution where software can run on a virtualized SoC model, like an Instruction Set Simulator or QEMU. At the same time, the hardware would be a MATLAB/Simulink model. Such an approach is presented [7]. Adopting the same kind of approach would enable testing of other features like data transfers, interrupts, inter-processor communications, and other peripheral configurations rather than just algorithmic correctness. This would also allow the integration and testing with other system components in an early project phase, even before RTL is available. This inexpensive and replicable virtual platform would be a beneficial step between subsystem-specific algorithm simulation environments, like MATLAB, and pre-silicon environments.

The suggested framework will bring extra benefit in helping to integrate the models and software because the Abstraction Layer API will hide the used environment from the software. Since the API and the model use the same parameters, there is also no manual mapping necessary. Parameters can be considered model registers, and no further mapping is needed besides some scripting. If the actual register interface is introduced later, it can be added without changing the software developed on top of the Abstraction Layer API.

Thorough documentation of the framework and workflow is needed as a further enhancement to make the adoption easier for new developers. Understanding the workflow's big picture is crucial as the model is a collaboration between different teams and forms a digital thread from algorithm design to final product. Avoiding dependencies also needs special attention. For example, having HW details like register content information in the model results in having cyclic dependency between the model and a specific HW version.

VII. CONCLUSION

Developing software for today's Digital Front End SoCs faces challenges regarding design validation software availability. This can be generalized as a problem for any large ASIC design, which can be characterized as algorithmically complex, highly configurable, and with many configuration parameters and hard time-to-market pressure. The tremendous effort needed for modelling, verification, and software implementation in a short time period before the design tape-out is the common denominator of these kinds of designs. Real production software for pre-silicon verification avoids wasting test scripting labor, enables more thorough testing, and improves software quality. It also reduces possibly finding costly bugs as the productization continues.

The described model-based reusable framework was introduced to enable early software development and fast deployment with the help of code generation. The development cycle was shortened as a result, and the hardware verification was completed using real production software for the first time. Apart from the software quality increase, due to verification shift-left, there were also other positive findings in the model-based design approach, namely in cross-team collaboration and communication. The framework's reusability will also make future product development faster and more predictable.

The described approach was applied to the Nokia 5G Digital Front End design. It can be used for any IP development that includes software and hardware complex enough to benefit from modelling. However, it is most beneficial for large SoC/ASIC designs where software implementation effort is so considerable that it is difficult to have a version mature enough for hardware pre-silicon verification.

A new virtual platform was proposed as a future enhancement. QEMU- or Instruction Set Simulator-based SoC/processor models could be integrated with existing Simulink models. This would provide an inexpensive and easily replicable environment for integrating generated software and different hardware models before proceeding to more expensive and reduced access to pre-silicon environments like emulators. The proposed framework would also benefit this new platform because of the direct mapping of the Abstraction Layer and model parameters.

REFERENCES

[1] D. Alagna, M Annovazzi, A. Cannone, M. Raimondi, S. Saracino, M. Chugh, M. Erickson, C. Macario, G. Ridinò, "*Reuse of System-Level Verification Components within Chip-Level UVM Environments*", DVCON Europe 2021, October 2021

[2] T. Fitzpatrick, P. Gupta, M. Vax, K. Gururaj, H. Miller, "*Portable Stimulus: What's Coming in 1.1 and What it Means For You*", DVCON U.S. 2020, March 2020

[3] M. Thanner, I. Feldner, S. Loitz, R. Schleifer, K. Brand, "*Automotive Virtual Prototypes*", DVCon Europe 2020, October 2020

[4] M. Burton et al., "*Hybrid System Simulation Standards*", DVCon Europe 2020, October 2020

[5] Axel Jahnke, "*Challenges in Soc Verification for 5G and Beyond*", DVCon Europe 2022, December 2022

[6] Barry W. Boehm, "*Software Engineering Economics*", Prentice Hall 1981, January 1981

[7] F. Poppen, K. Grüttner , "*Co-Simulation of C-Based SoC Simulators and Matlab Simulink*", Proceedings of the Operational Research Society Simulation Workshop 2012