Efficient Techniques to Strongly Enhance the Virtual Prototype based Design Flow*

Vladimir Herdt Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{vherdt,drechsle}@informatik.uni-bremen.de

***Abstract*—SystemC-based Virtual Prototypes (VPs) are an industry-proven solution to tackle the rising complexity of embedded systems in the design flow. This paper proposes a comprehensive set of novel approaches that strongly enhance all major aspects of a modern VP-based design flow. A strong emphasis is put on automated formal verification methods and advanced coverage-guided testing techniques tailored for SystemC-based VPs and also the software. In addition, we consider VP modeling techniques that cover functional as well as non-functional aspects and also propose automated correspondence analyses between the hardware- and VP-level to utilize information available at different levels of abstraction. All approaches have been extensively evaluated with several experiments that clearly demonstrate their effectiveness in strongly enhancing the VP-based design flow, in particular by drastically improving the overall quality in combination with a reduction in time-to-market. Furthermore, this paper puts a particular focus on the modern RISC-V *Instruction Set Architecture* (ISA).**

I. Introduction

Embedded systems are prevalent nowadays in many different application areas ranging from *Internet-of-Things* (IoT) to automotive and production as well as communication and multi-media applications. Embedded systems consist of *Hardware* (HW) and *Software* (SW) components and are typically small resource constrained systems that are highly specialized to implement application specific solutions. Hence, design flows for embedded systems require efficient and flexible design space exploration techniques to satisfy all application specific requirements such as power consumption and performance constraints.

To cope with the rising complexity of embedded devices, a *Virtual Prototype* (VP) based design flow is being widely adopted [1]–[8]. A VP is essentially an executable abstract model of the entire *Hardware* (HW) platform and pre-dominantly created in SystemC TLM (*Transaction Level Modeling*) [9], [10]. In contrast to a traditional design flow, which first builds the HW and then the *Software* (SW), a VP-based design flow enables parallel development of HW and SW by leveraging the VP for early SW development and as reference model for the subsequent design flow steps. However, this modern VP-based design flow still has weaknesses, in particular due to the significant manual effort involved for verification and analysis as well as modeling tasks which is both time consuming and error prone.

This paper proposes several novel approaches that cover modeling, verification and analysis aspects to strongly enhance the VP-based design flow. It is a summary of the PhD thesis [11]. The contributions are essentially divided in four areas: The first contribution is an open-source RISC-V VP that is implemented in SystemC TLM and covers functional as well as non-functional aspects [12]–[14]. The second contribution improves the verification flow for VPs by considering novel formal verification methods and advanced automated coverage-guided testing techniques tailored for SystemC-based VPs [15]–[21]. The third contribution are efficient coverage-guided approaches that improve the VP-based SW verification and analysis, covering functional as well as non-functional aspects [22]–[25]. The fourth and final contribution are approaches that perform a correspondence analysis between RTL (*Register-Transfer Level*) and TLM to utilize information available at different levels of abstraction [26], [27]. All approaches have been extensively evaluated with several experiments that clearly demonstrate their effectiveness in strongly enhancing the VP-based design flow. In the following we summarize the main results of the four contribution areas.

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONFIRM under contract no. 16ES0565 and within the project EffektiV under contract no. 01IS13022E and within the project VerSys under contract no. 01IW19001.

II. Open-Source RISC-V Evaluation Platform

The first contribution is an open-source RISC-V VP implemented in SystemC TLM. The VP serves as evaluation platform for several VP-based approaches.

RISC-V is an open and free *Instruction Set Architecture* (ISA) [28], [29] which is license-free and royalty-free. Similar to the enormous momentum of open-source SW the open-source RISC-V ISA is also gaining large momentum in both industry and academia. In particular for embedded devices, e.g. in the IoT area, RISC-V is becoming a game changer. A large and continuously growing ecosystem is available around RISC-V ranging from several HW implementations (i.e. RISC-V cores) to SW libraries, operating systems, compilers and language implementations. In addition, several open-source high-speed *Instruction Set Simulators* (ISS) are available. However, these ISSs are primarily designed for a high simulation performance and hence can hardly be extended to support further system-level use cases such as design space exploration, power/timing/performance validation or analysis of complex HW/SW interactions. The goal of the proposed RISC-V VP is to fill this gap in the RISC-V ecosystem and stimulate further research and development.

The VP provides a 32/64 bit RISC-V core with an essential set of peripherals and support for multi-core simulations. In addition, the VP also provides SW debug (through the Eclipse IDE) and coverage measurement capabilities and supports the FreeRTOS, Zephyr and Linux operating system. The VP is designed as extensible and configurable platform (as an example we provide a configuration matching the RISC-V HiFive1 board from SiFive) with a generic bus system and implemented in standard-compliant SystemC. The latter point is very important, since it allows to leverage cutting-edge SystemC-based modeling techniques needed for the mentioned system-level use cases. Finally, the VP allows a significantly faster simulation compared to RTL, while being more accurate than existing ISSs.

In addition, the VP integrates an efficient core timing model to enable fast and accurate performance evaluation for RISC-V based systems. The timing model is attached to the core using a set of well-defined interfaces that decouple the functional from the non-functional aspects and enable easy re-configuration of the timing model. As example a timing configuration matching the RISC-V HiFive1 board from SiFive is provided.

III. Verification of SystemC-based VPs

By virtue of their impact on the design flow, verification of VPs is crucial. We consider novel formal verification methods as well as advanced coverage-guided test-case generation techniques, which are presented in the following.

A. Formal Verification using Symbolic Simulation

Formal verification of SystemC is very challenging due to its object-oriented nature and event-driven simulation semantics [30]:

1) It must obviously consider all possible inputs of the *Design-Under-Verification* (DUV).
2) A typical high-level SystemC DUV consists of multiple asynchronous processes, whose different orders of execution (i.e. *schedules*) can lead to different behaviors, these must also be considered to the full extent by the verifier.
3) The defined state space very often contains cycles that arise naturally due to the use of unbounded loops inside the asynchronous processes.
4) The verifier is required to deal with the full complexity of C++ to extract a suitable formal model.

A promising direction to make the overall challenge more manageable is to work on an *Intermediate Verification Language* (IVL) that separates front-end and back-end issues. Such an IVL for SystemC has been introduced in our previous work [31]. The IVL is an open, compact and readable language with a freely available parser designed for both manual and automatic transformations from SystemC.

a) Stateful Symbolic Simulation: With the IVL in place one can focus on developing techniques to enhance the scalability and efficiency of the back-end (i.e. addressing the first three challenges). To this end, we present a stateful symbolic simulation approach together with a *State Subsumption Reduction* (SSR) technique at its heart, for the efficient verification of cyclic state spaces in high-level SystemC designs. More precisely, the stateful symbolic simulation combines SSR with POR and *Symbolic Execution* (SymEx) [32] under the SystemC simulation semantics. POR prunes redundant process scheduling sequences, while SymEx efficiently explores all conditional execution paths of each individual process in conjunction with symbolic inputs. Subsequently, the SymEx+POR combination enables to cover all possible inputs and scheduling sequences of the DUV exhaustively in a stateless manner. To deal with cycles, the stateful search keeps a record of already visited states to avoid re-exploration.

However, before discussing SSR, please let us note that two major challenges must be solved to enable a stateful search in combination with SymEx and POR. First, SymEx stores and manipulates symbolic expressions, which represent sets of concrete values. Therefore, the *state matching* process, required by a stateful search to decide whether a state has already been visited, involves non-trivial comparison of complex symbolic expressions. Second, a naive combination of POR with stateful search can potentially lead to unsoundness, i.e. assertion violations can be missed. This is due to the (transition) *ignoring problem*, which refers to a situation, where a relevant transition is not explored.

SSR solves the first challenge by applying *symbolic subsumption checking*, inspired by [33]. The basic idea is as follows. If the set of concrete states represented by a symbolic state s_2 contains the set of concrete states represented by a symbolic state s_1, s_1 is *subsumed* by s_2 and it is not necessary to explore s_1 if s_2 has already been explored. We employ a powerful exact subsumption checking method which involves solving a quantified SMT formula. As this computation is potentially very expensive, we also employ several optimizations. To address the second issue, we develop a tailored *cycle proviso* and show that its integration preserves the soundness of our stateful symbolic simulation (i.e. SSR+SymEx+POR combination).

We have implemented the techniques in a tool named SISSI (**S**ystemC **I**VL **S**ymbolic **Si**mulator). Our preliminary experiments [31] have already shown the potential of the SymEx+POR combination in comparison with all available state-of-the-art formal approaches. The experiments focus on demonstrating the efficiency of the final stateful approach using an extensive set of benchmarks.

b) Experiments: We have implemented the presented stateful symbolic simulation approach in Python (version 3.6.0) and evaluated it using the extensive set of benchmarks available in the IVL format. All experiments are performed on a 3.5 GHz Intel machine running Linux. The time and memory limits are set to 1000 seconds and 6GB, respectively. The abbreviations T.O. and M.O. denote that the time and memory limit has been exceeded, respectively. N.S. denotes an unsupported benchmark. For unsafe benchmarks the verifier will stop once the first bug has been found, for safe benchmarks it needs to prove correctness. Every benchmark contains a cyclic state space. The Z3 solver [34] in version 4.5.0 is used to handle all symbolic queries, since it provides quantifier support as required for the SSR algorithm. We compare our tool SISSI against KRATOS [35] the state-of-the-art model checker for SystemC for handling cyclic state spaces.

Table I shows the results. It shows the benchmark name in the first column and the verification result (V), with S for safe and U for unsafe, in the second column. The third (*#IVL*) and fourth column (*#P*) shows the lines of code in IVL and the number of processes, respectively. All runtimes are specified in seconds.

Our approach shows very competitive results compared to KRATOS. Improvements up to two orders of magnitude can be observed. This can especially be observed with up-scaled benchmarks, e.g. the *token-ring*, *transmitter* and *pressure* benchmarks. This also demonstrates the scalability of our approach. For example, (nearly) doubling the number of processes from 51 to 101 in the transmitter benchmark does increase the runtime of our tool by 2.6x, whereas the runtime of KRATOS is increased by 58.1x. On some benchmarks, KRATOS shows better results but the runtime differences are not significant. This can be explained as follows. KRATOS starts with a coarse abstraction of the design and gradually refines it until a (real) counter-example is detected or safety is proven on the (conservative) abstraction. This is a complete approach that works very well for benchmarks that require only a small number of refinement steps. Furthermore, our approach is implemented in an interpreted language and thus might have a larger overhead on easy benchmarks. Also, on some safe benchmarks, KRATOS reports spurious counterexamples. These benchmarks are marked with *.

c) Optimization and Application: In [17] we proposed Compiled Symbolic Simulation (CSS) to further boost scalability. CSS is a major enhancement that augments the DUV to integrate the symbolic execution engine and the POR based scheduler. Then, a C++ compiler is used to generate a native binary, whose execution performs exhaustive verification of the DUV. The whole state space

TABLE I
COMPARISON WITH KRATOS (RUNTIME IN SECONDS)

Benchmark	V	#IVL	#P	KRATOS	**SISSI**
buffer.p8	S	99	9	23.68	6.49
buffer.p9	S	107	10	526.78	14.46
pc-sfifo-sym-1	S	49	2	0.14	0.50
pc-sfifo-sym-2	S	65	2	0.17	0.53
pressure2-sym.nb.10.5	S	52	5	1.25*	12.23
pressure2-sym.nb.50.5	S	52	5	153.30*	158.38
pressure-sym.nb.10.5	S	33	3	0.48*	5.58
pressure-sym.nb.50.5	S	33	3	51.90*	70.60
rbuf-2	S	82	3	15.58*	0.67
rbuf-4	S	108	5	16.04*	0.79
simple-fifo-1c2p.20	S	83	3	37.03	1.59
simple-fifo-1c2p.50	S	83	3	T.O.	3.99
simple-fifo-2c1p.20	S	83	3	33.47	2.23
simple-fifo-2c1p.50	S	83	3	T.O.	6.07
simple-fifo-bug-1c2p.20	U	79	3	18.28	1.06
simple-fifo-bug-2c1p.20	U	79	3	14.68	1.17
symbolic-counter.1.15	S	41	3	559.96	5.00
symbolic-counter.9.15	S	41	3	155.02	2.06
token-ring2.12	S	224	13	42.62	39.00
token-ring2.20	S	360	21	M.O	229.43
token-ring-bug2.17	U	308	18	22.84	1.76
token-ring-bug2.50	U	869	51	M.O	5.14
token-ring-bug.20	U	220	21	133.20	1.95
token-ring-bug.40	U	420	41	M.O	3.58
token-ring.13	S	150	14	2.02	2.55
token-ring.15	S	170	16	32.09	3.20
token-ring.40	S	420	41	M.O	48.57
toy-sym	S	86	5	0.46	1.56
transmitter.50	U	414	51	1.13	1.60
transmitter.90	U	734	91	M.O	3.54
transmitter.100	U	814	101	65.69	4.21

of a DUV, which consists of all possible inputs and process schedules, can thus be exhaustively and efficiently explored. To further boost the verification process, we proposed an efficient parallelized exploration algorithm in [18].

In [16] we first showed how to bridge the modeling gap between the industry-accepted modeling pattern for TLM peripherals and the semantics currently supported by SystemC formal verification approaches. Then, we reported verification results for the interrupt controller of the LEON3-based SoCRocket VP [3] used by the European Space Agency.

B. Advanced Coverage-guided Testing Techniques

Despite the recent progress in formal verification of SystemC designs as presented in the previous section, simulation-based verification is still the method of choice for SystemC-based VPs in industrial practice thanks to its scalability and ease of use. Basically, a set of stimuli is applied to the *Design Under Verification* (DUV; which can be either a whole VP, a set of components or a single component) and for each stimulus, the actual behavior is checked against the expected behavior (e.g. specified by reference outputs or temporal properties). Since a VP is in essence a software model, simulation-based verification for VPs is actually very similar to software testing and therefore, techniques from the software domain can be borrowed to ensure a high quality of verification results. For example, high statement coverage of the DUV implied by the set of stimuli (also referred to as *test-suite* in the following to reflect the software testing point of view) is nowadays a minimum requirement. This

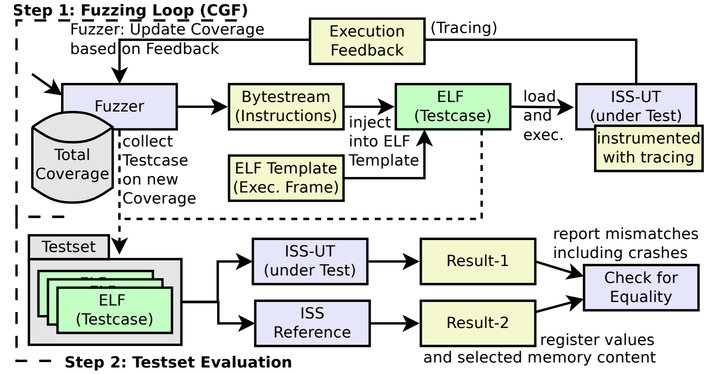

Fig. 1. Overview on CGF for ISS verification

section dicusses two approaches for VP verification using coverage-guided testcase generation. Both approaches have been shown very successful in the SW domain to generate a comprehensive testsuite with a high error detection rate.

The first approach is *Data Flow Testing* (DFT) for SystemC-based VPs [20]. The contribution is twofold: First, we developed a set of SystemC specific coverage criteria for DFT. This requires to consider the SystemC semantics of using non-preemptive thread scheduling with shared memory communication and event-based synchronization. Second, we explained how to automatically compute the data flow coverage result for a given VP using a combination of static and dynamic analysis techniques. The coverage result provides clear suggestions for the testing engineer to add new testcases in order to improve the coverage result. The experimental results on real-world VPs demonstrated the applicability and efficacy of the analysis approach and the SystemC specific coverage criteria to improve the testsuite.

The second approach leverages *Coverage-Guided Fuzzing* (CGF) to improve the testcase generation process for verification of *Instruction Set Simulators* (ISS) [21], which is a crucial component in every VP. In addition to code coverage we integrated functional coverage and a custom mutation procedure tailored for ISS verification. Fuzzing is particularly useful to trigger and check for error-cases and can complement other testcase generation techniques. As a case-study we applied our approach on a set of three publicly available RISC-V ISSs. We found several errors, including one error in the official RISC-V reference simulator *Spike*. We present an overview and the main results in the following.

a) CGF for ISS Verification: Fig. 1 shows an overview on our approach for ISS verification. Essentially, it consists of two subsequent steps: First a testset is generated by the fuzzer (upper half of Fig. 1), then the testset is used to verify the functionality of the ISS under test (ISS-UT, lower half of Fig. 1) by comparing the execution results with other reference ISSs (can be multiple). In the following we describe the fuzzer loop in more detail.

The fuzzer starts with an empty coverage and empty testset. The fuzzer iterates until the coverage goal or the specified time limit is reached. In each step the fuzzer generates a (binary) bytestream. We interpret this bytestream as a sequence of instructions for the ISS under test (ISS-UT). Every such bytestream is transformed into an (ELF-)testcase, by embedding the bytestream into a pre-defined ELF-template[1].

[1]Technically, we use a linker script that generates an empty section in the ELF-template. Then we use *objcopy* utility with the *–update-section* argument to overwrite the empty section with the (binary) instruction bytestream.

TABLE II
EVALUATION RESULTS - ALL EXECUTION TIMES ARE REPORTED IN SECONDS - [V1..V7] MEANS ALL 7 ERRORS V1 TO V7 HAVE BEEN FOUND.

Tests / Generator	Time	Coverage		Errors found in ISS		
	(sec.)	Code	Func.	UT	Spike	Forvis
T1: ISA Tests	2	90.24%	37.89%	[V1..V3]	/	/
T2.1: Torture 1000	5280	74.30%	56.20%	V1,V2	/	H2
T2.2: 5000	26108	74.30%	57.88%	V1,V2	/	H2
T2.3: 10000	52168	74.30%	63.56%	V1,V2	/	H2
T3: CGF	32492	100.00%	97.42%	[V1..V7]	S1	H1,H2

The ELF-template contains prefix and suffix code (execution frame) that is supposed to be executed before and after the actual sequence of instructions. The prefix is responsible to initialize the ISS into a pre-defined initial state. This includes initializing all registers to pre-defined values to ensure that all ISS implementations start in the same state. The suffix is responsible to collect results and stop the simulation. It will write all register values into a pre-defined region in memory (can contain additional content beside the register values) to enable dumping the result of the execution to a file (an ISS typically provides an operation to dump specific memory regions), which can be compared.

The testcase is then executed on the ISS-UT. The ISS-UT generates execution feedback by tracing relevant information. The tracing functionality need to be instrumented into the ISS-UT. The fuzzer will analyze the execution feedback and update its coverage metrics accordingly. In case the coverage is increased by executing the testcase, the testcase is added to the fuzzer testset.

b) Experiments: RISC-V ISS Verification: As a case study we built our CGF approach on top of *libFuzzer* and verified the RISC-V ISS extracted from our open-source RISC-V Virtual Prototype (VP) [36]. We denote this ISS as *ISS-UT*. As reference we use the following two ISS:

1) *Spike*, the official RISC-V ISA reference simulator [37].
2) *Forvis*, an ISS implemented in Haskell aiming to be a formal specification of the RISC-V ISA [38].

Table II shows the results. The table is separated into four main columns. The first column (Tests/Generator) reports which testset is used or how it has been obtained, respectively. In addition to our CGF, we also use the official RISC-V ISA tests [39] and the RISC-V Torture testcase generator [40] in the comparison to further evaluate the effectiveness of our fuzzing approach. The second column (Time) shows the time in seconds to generate (9h timeout for our fuzzer) and execute the respective testcases. Please note, the RISC-V ISA tests are directed tests that are hand-written and thus do not require a generation step. The third column shows the code- (branch coverage in particular) and functional coverage obtained by running the respective testset. The coverage is measured based on the instrumented ISS-UT. The fourth and last column shows which (and how many) errors have been found by each approach.

Our fuzzing-based approach CGF is able to detect all errors found by the ISA tests as well as Torture test generator and finds six additional errors (4 in ISS-UT, 1 in Spike and 1 in Forvis), see (Table II, column *"Errors found in ISS"*, row T3). Most of these errors relate to dealing with different forms of illegal instructions in different steps of the execution process. Besides being coverage-guided, a major benefit of our fuzzer is being not constrained to some specific instruction subset, as for example Torture (and hence could not detect the errors our fuzzer did, independent of the number of testcases generated). In particular the fuzzer operates on the binary level, thus it can be used to check for errors that might even be masked by a compiler/assembler (as they do not generate illegal instructions). This also enables to thoroughly check the instruction decoder unit, which even revealed an error in the RISC-V reference simulator Spike.

IV. VP-BASED SW VERIFICATION AND ANALYSIS

After verifiying the VP, the VP is used as platform for SW development. Todays SW is becoming increasingly complex and encompasses several abstraction layers ranging from bootcode and device drivers to complex libraries and operating systems. Verification of the embedded SW is very important to avoid errors and security vulnerabilities.

In addition, modern systems must satisfy stringent requirements on power consumption and performance. With a continuously fast increase in number of implemented functionalities as well as in their complexity, meeting these requirements has become one of the major challenges in embedded system design. This new challenge demands a major shift in the design flow where power optimization/management is no longer an afterthought but considered early at the system level. Here, the focus is not on low-level techniques such as power gating or dynamic voltage and frequency scaling but rather on fundamental design decisions that have a large impact on the power consumption such as power management strategies. These strategies are typically implemented in firmware and can contribute a great deal to the overall power saving by putting unused components into low-power states and waking them up properly in an intelligent manner. Validation of these firmware-based strategies is very important.

In the following we present our VP-based verification and validation approaches in more detail.

A. Verification of Embedded Software Binaries using VPs

Here we proposed two approaches. The first approach leverages state-of-the-art CGF methods in combination with SystemC-based VPs for verification of embedded SW binaries [23]. Using VPs, the approach allows a fast and accurate binary-level SW analysis and enables checking of complex HW/SW interactions. To guide the fuzzing process the coverage from the embedded SW is combined with the coverage of the SystemC-based peripherals. The experiments, using real-world RISC-V embedded SW binaries as examples, demonstrate the effectiveness of the proposed approach. For example, we obtained a high coverage by fuzzing the TCP/IP stack of the Zephyr operating system. The analyzed RISC-V binary had 46,105 lines of assembly code.

The second approach leverages concolic testing tailored for binaries targeting RISC-V systems with peripherals [22]. The approach works by integrating the *Concolic Testing Engine* (CTE) with the architecture specific *Instruction Set Simulator* (ISS) inside of a VP. A designated CTE-interface is provided to integrate (SystemC-based) peripherals into the concolic testing by means of SW models. This combination enables a high simulation performance at binary level with comparatively little effort to integrate peripherals with concolic execution capabilities. We applied our approach to analyze the TCP/IP stack of FreeRTOS (v10.0.0) in combination with the RISC-V port of the FreeRTOS kernel. Therefore, we essentially injected a single (small) packet with symbolic size and content into the TCP stack and checked for generic execution errors (including FreeRTOS assertions) and heap buffer overflows. We found several buffer overflow based security vulnerabilities (the runtime was around 860 seconds, our tool executed around 2.5 billion RISC-V instructions and checked around 14,500 symbolic execution paths).

B. Validation of FW-based Power Management using VPs

The recent advances in ESL power modeling and estimation enable to execute a particular SW application in FW/VP co-simulation and check whether its power budget and performance requirement are met. However, there is still a number of shortcomings with this basic simulation-based approach. First, production-level SW is not yet available in early design stages. Second, simulating a full SW stack can still be very time-consuming, even at the speed of VPs. Third, a SW application is executed under some predetermined workloads (i.e. application and environment inputs). These workloads might very possibly miss rare corner cases where the power budget is exceeded or the performance constraint is violated.

To address these shortcomings, we propose a novel VP-based approach to assess the power-versus-performance trade-off of FW-based power management [24]. Instead of executing real SW applications, the approach makes use of system-level workload scenarios. The main novelty of the approach is the modeling of workload scenarios based on *constrained random* (CR) techniques [41] that are very successful in the area of SoC/HW functional validation and verification. Each workload scenario corresponds to a system-level use-case with a specific power consumption profile and is described by a set of constraints. The constraints define the set of legal concrete workloads that are conform to the intended use-case. The constraint-based description enables automated generation of a large number of different workloads within the scenario, hence reducing the risk of missing a corner case.

Although the constraint-based workload description enables automated generation of a large number of different test-cases (corresponding to SW workloads), hence reducing the risk of missing a corner case, a coverage metric to objectively measure the quality aspect as well as to guide the generation of scenarios is still an important missing piece. At the very least, it is mandatory that all power states of each component are comprehensively exercised by the generated test-cases. Recent experience from the industry [42] makes a case for using stronger metrics. This paper argues that the power states from different components or power domains are not necessarily independent. This also applies to the context of FW-based PM, since this global management scheme can change the power state of several components simultaneously according to the implemented strategy. Therefore, an appropriate coverage metric must account for all possible combinations of these interdependent power states. The cross coverage of power states is such a metric.

Thus, we propose a novel coverage-driven validation approach for FW-based PM [25]. The main contribution is a feedback-directed workload generation algorithm that generates testcases in an automated manner in order to maximize the cross coverage of power states. The approach works in two phases: first a bootstrap phase is performed to obtain preliminary coverage information based on randomly generated test-cases and then a coverage-loop phase to close the remaining coverage gaps. The main idea is to mix different pre-calibrated SW blocks, that represent abstract workloads, in order to move the system into specific power states. Fig. 2 illustrates that idea: by mixing two SW blocks, the overall system load moves on a line between these blocks. A refinement loop is integrated to further guide the test-case generation process (adapt how exactly the blocks are mixed by observing their real effects to reach a specific goal load that triggers a power state transition). The applicability and efficiency of the proposed approach in generating SW to obtain a high power-state cross-coverage is demonstrated using four different PM strategies.

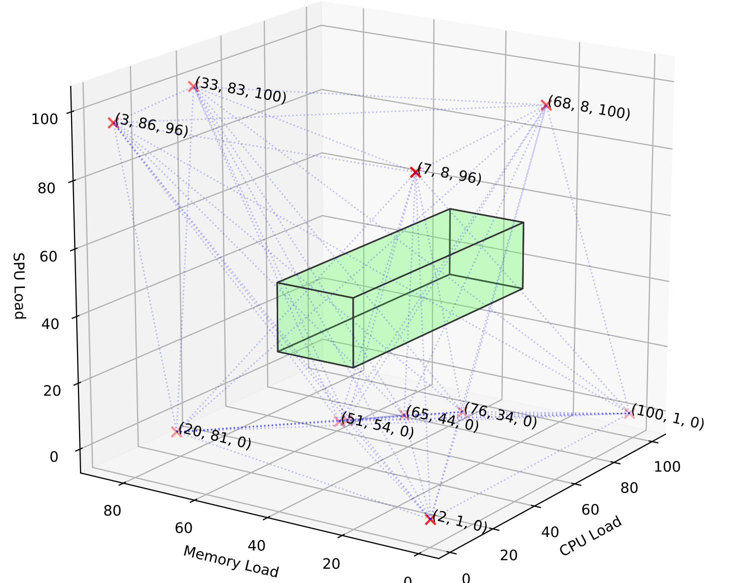

Fig. 2. Example goal load cuboid (three dimensional load vector) together with the eleven SW block load vectors and all line combinations (dashed blue) shown in the cross load state space of a CPU, memory and SPU component.

V. RTL Correspondence Analysis

The final contribution of this work are two approaches that perform a correspondence analysis between TLM and RTL.

The first proposed approach enables an automated TLM-to-RTL property refinement [27]. It enables to transform high-level TLM properties into RTL properties to serve as starting point for RTL property checking. This avoids the manual transformation of properties from TLM to RTL which is error prone and time consuming.

The second proposed approach performs an RTL-to-TLM fault correspondence analysis [26]. It enables to identify corresponding TLM errors for transient bit flips at RTL. The obtained results can improve the accuracy of a VP-based error effect simulation. An error effect simulation essentially works by injecting errors into the VP during SW execution to check the robustness of the SW against different HW faults. Such an analysis is very important for embedded systems that operate in vulnerable environments or perform safety critical tasks to protect against effects of for example radiation and aging.

VI. Going Beyond the VP-based Design Flow

We considered additional verification and analysis aspects which are related but go beyond the general VP-based design flow, which we briefly summarize in the following. In [43] we proposed a novel approach to SoC security validation at the system level using VPs. We proposed a novel resilience evaluation framework combining LLVM-based SW fault injection and SMT-based symbolic execution in [44]. In [45], [46] we presented an optimized symbolic verification technique for concurrent C programs using POSIX threads. [47] discusses methods to enable an early, efficient and systematic design of FW. Finally, in [48], [49] we proposed techniques for RISC-V compliance testing that cover positive and negative testing aspects.

VII. Conclusion

In the last years the complexity of embedded devices has been increasing steadily with various conflicting requirements. On the one hand IoT devices need to provide smart functions with a high performance including real-time computing capabilities, connectivity and

remote access as well as safety, security and high reliability. At the same time they have to be cheap, work efficiently with an extremely small amount of memory and limited resources and should further consume only a minimal amount of power to ensure a very long runtime.

To cope with the rising design complexity a VP-based design flow is employed. In contrast to a traditional design flow which first builds the HW and then the SW, a VP-based design flow enables HW/SW Co-Design by leveraging the VP for early SW development and as reference model for the subsequent design flow steps. However, this modern VP-based design flow still has weaknesses, in particular due to the significant manual effort involved for verification and analysis tasks which is both time consuming and error prone. We proposed several novel approaches to strongly enhance the VP-based design flow and with this provides contributions to every main step in a VP-based design flow.

All approaches have been extensively evaluated with several experiments. In summary, these contributions significantly enhance the VP-based design flow by drastically improving the verification quality and at the same time reduce the overall verification effort due to the extensive automatization.

REFERENCES

[1] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[2] R. Leupers, F. Schirrmeister, G. Martin, T. Kogel, R. Plyaskin, A. Herkersdorf, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *DATE*, 2012, pp. 685–690.

[3] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "Socrocket - A virtual platform for the European Space Agency's SoC development," in *ReCoSoC*, 2014, pp. 1–7, available at http://github.com/socrocket.

[4] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, "Fast virtual prototyping for embedded computing systems design and exploration," in *RAPIDO Workshop*, 2019, pp. 3:1–3:8.

[5] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh, "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems," in *DATE*, 2015, pp. 1698–1707.

[6] J. H. Oetjens, N. Bannow, M. Becker, O. Bringmann, A. Burger, M. Chaari, S. Chakraborty, R. Drechsler, W. Ecker, K. Grüttner, T. Kruse, C. Kuznik, H. M. Le, A. Mauderer, W. Müller, D. Müller-Gritschneder, F. Poppen, H. Post, S. Reiter, W. Rosenstiel, S. Roth, U. Schlichtmann, A. von Schwerin, B. A. Tabacaru, and A. Viehl, "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *DAC*, 2014, pp. 1–6.

[7] J. Kong, B. Yoo, D. Song, H. J. Nam, J. Hwang, J. Kim, S. Lee, S. Eo, S. Yoo, K. Choi, H. Jin, J. Kim, S. Lee, and S. Hong, "Creation and utilization of a virtual platform for embedded software optimization:: an industrial case study," in *CODES+ISSS*, 2006, pp. 235–240.

[8] A. Kramer and M. Vaupel, "Virtual platforms for automotive: Use cases, benefits and challenges," https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2014/proceedings/T01_tutorial_part3.pdf, 2014.

[9] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[10] *OSCI TLM-2.0 Language Reference Manual*, OSCI, 2009.

[11] V. Herdt, "Efficient modeling, verification and analysis techniques to enhance the virtual prototype based design flow for embedded systems," Ph.D. dissertation, University of Bremen, 2020.

[12] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.

[13] V. Herdt, D. Große, and R. Drechsler, "Fast and accurate performance evaluation for RISC-V using virtual prototypes," in *DATE*, 2020.

[14] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," in *JSA*, 2020.

[15] V. Herdt, H. M. Le, and R. Drechsler, "Verifying SystemC using stateful symbolic simulation," in *DAC*, 2015, pp. 49:1–49:6.

[16] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models – a case study," in *DATE*, 2016, pp. 1160–1163.

[17] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.

[18] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "ParCoSS: efficient parallelized compiled symbolic simulation," in *CAV*, 2016, pp. 177–183.

[19] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, vol. 38, no. 7, pp. 1359–1372, July 2019.

[20] M. Hassan, V. Herdt, H. M. Le, M. Chen, D. Große, and R. Drechsler, "Data flow testing for virtual prototypes," in *DATE*, 2017, pp. 380–385.

[21] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.

[22] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019, pp. 188:1–188:6.

[23] V. Herdt, D. Große, J. Wloka, T. Güneysu, and R. Drechsler, "Verification of embedded binaries using coverage-guided fuzzing with SystemC-based virtual prototypes," in *GLSVLSI*, 2020.

[24] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, 2017, pp. 1–8.

[25] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Maximizing power state cross coverage in firmware-based power management," in *ASP-DAC*, 2019, pp. 335–340.

[26] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate VP-based error effect simulation - a case study," in *FDL*, 2016, pp. 1–8.

[27] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards fully automated TLM-to-RTL property refinement," in *DATE*, 2018, pp. 1508–1511.

[28] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[29] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[30] M. Y. Vardi, "Formal techniques for SystemC verification," in *DAC*, 2007, pp. 188–192.

[31] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *DAC*, 2013, pp. 116:1–116:6.

[32] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[33] S. Anand, C. S. Păsăreanu, and W. Visser, "Symbolic execution with abstract subsumption checking," in *SPIN*, 2006, pp. 163–181.

[34] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340, available at https://github.com/Z3Prover/z3.

[35] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *TCAD*, vol. 32, no. 5, pp. 774–787, 2013.

[36] "RISC-V virtual prototype," https://github.com/agra-uni-bremen/riscv-vp.

[37] "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim.

[38] "Forvis: A formal RISC-V ISA specification," https://github.com/rsnikhil/RISCV-ISA-Spec.

[39] "RISC-V ISA tests," https://github.com/riscv/riscv-tests.

[40] "RISC-V torture test generator," https://github.com/ucb-bar/riscv-torture.

[41] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.

[42] V. V. Singh and A. Kumar, "Cross coverage of power states," in *DVCon*, 2016.

[43] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Early SoC security validation by VP-based static information flow analysis," in *ICCAD*, 2017, pp. 400–407.

[44] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Resiliency evaluation via symbolic fault injection on intermediate code," in *DATE*, 2018, pp. 845–850.

[45] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Boosting sequentialization-based verification of multi-threaded C programs via symbolic pruning of redundant schedules," in *ATVA*, 2015, pp. 228–233.

[46] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Combining sequentialization-based verification of multi-threaded C programs with symbolic partial order reduction," *STTT*, vol. 21, no. 5, pp. 545–565, 2019.

[47] V. Herdt, D. Große, R. Drechsler, C. Gerum, A. Jung, J.-J. Benz, O. Bringmann, M. Schwarz, D. Stoffel, and W. Kunz, "Systematic RISC-V based firmware design," in *FDL*, 2019, pp. 1–8.

[48] V. Herdt, D. Große, and R. Drechsler, "Towards specification and testing of RISC-V ISA compliance," in *DATE*, 2020.

[49] V. Herdt, D. Große, and R. Drechsler, "Closing the RISC-V compliance gap: Looking from the negative testing side," in *DAC*, 2020.