Exploring the Parameter Space for Constrained Random Verification of RISC-V CPUs

Sallar Ahmadi-Pour[1], Luca Müller[2], Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{sallar, lucam, drechsler}@uni-bremen.de

***Abstract*—With the growing complexity of modern *Integrated Circuits (ICs)*, cross-level verification increasingly relies on multiple coverage metrics to assess verification progress and quality. While *Virtual Prototypes (VPs)* enable efficient verification across abstraction layers, the impact of verification parameters on bug-finding effectiveness remains poorly understood. This work presents an exploratory analysis of verification parameters in a *Constrained Random Verification (CRV)*-based cross-level setup. Using a RISC-V *Register-Transfer Level (RTL)* implementation and a binary-compatible VP, we systematically explore a parameterized CRV-based verification flow, varying instruction sequence length, mutation location, and mutation count, and assess their impact using mutation, functional, and code coverage metrics. Our results show that different coverage metrics exhibit distinct saturation behaviors as verification parameters are varied. In particular, mutation coverage continues to reveal bug-finding effectiveness in scenarios where functional and code coverage stabilize, highlighting the importance of cross-checking trends across multiple metrics rather than maximizing individual coverage values in isolation.**

***Index Terms*—Verification, Constrained Random Verification, Cross-Level, Virtual Prototype, Register-transfer Level**

I. Introduction

Increasing IC complexity makes verification across abstraction levels increasingly important [1]–[3]. VPs provide fast, binary-compatible executable models that can serve as early system-level references for RTL verification [4], [5]. At the system-level, VPs have shown immense capabilities, including but not limited to testing and verification of *Software (SW)* [6], [7], functional verification [8], [9], or evaluation of platform and environment interaction [10]. In VP-RTL cross-level verification, the same SW binary runs on both models, and their traces are compared to spot behavioral mismatches. While promising, these flows' effectiveness depends on verification parameters such as program length and mutation location.

Recent works often utilize established methods from the domain of SW verification or leverage machine learning techniques to address scalability or coverage closure but ignore the various parameters available in the verification setup [11]–[13]. Moreover, a popular technique to quantify the bug-finding capabilities is to introduce artificial bugs through mutation of the original *Device under Verification (DUV)*. For VP-RTL cross-level verification, this spans a space for verification parameters (e.g., number of generated inputs for tests, or mutation location), where understanding the impact of such parameters is crucial for the quality of verification results. With verification across abstraction layers using VPs and coverage-driven, parameterized tests, it is essential to understand how verification parameters affect bug-finding effectiveness and coverage metrics.

To tackle this, we examine how verification parameters affect bug-finding effectiveness and coverage behavior in a cross-level CRV setup. Rather than pursuing coverage closure or parameter optimization, we aim to systematically characterize how parameters such as instruction sequence length, and mutation location affect different coverage metrics. In this context, mutation testing is used as a practical proxy for bug-finding effectiveness, capturing a broad range of fault types, including small design errors that may arise from specification misunderstandings or mistakes, while complementary coverage metrics provide additional perspectives on verification progress. While the presented results are obtained from a RISC-V processor case study, the design combines control-intensive and datapath-intensive components, making the derived insights applicable to a wider class of processor-like architectures.

1) Propose a parameter-centric framework for evaluating the impact of verification parameters with coverage-based and mutation-based assessment in CRV-based cross-level verification using mutation, functional, and code coverage metrics.
2) Systematically explore how the verification parameter space, such as instruction sequence length, mutation count, and mutation location influence bug-finding effectiveness and coverage behavior in a RISC-V VP-RTL cross-level setup, and derive insights on saturation and sensitivity of commonly utilized coverage metrics.
3) As part of the framework, we open-source the utilized scripts to automatically generate and inject bugs into RTL designs and their submodules, allowing fine-grained control over the location of injected mutations.[1]

This research has been supported by the German Ministry for Research, Technology and Space (BMFTR) with projects Scale4Edge (grant no. 16ME0127), ECXLplus (grant no. 01IW24001), and ExaVerse (grant no. 16IW25003).

[1]https://github.com/agra-uni-bremen/fdl-2026-mutationGeneration

II. Background and Related Work

A. Constrained Random Verification

The goal of *Hardware (HW)* verification, to ensure correctness of a DUV and covering as many behavioral aspects as possible, comes with various challenges. Different circuits can focus mostly on data flow or control flow, respectively, while other circuits might describe programmable and modular architectures with various interconnection schemes. Hence, challenges, such as state space explosion [14] or the programmable nature of a DUV, can lead to a differing effectiveness of existing and emerging methodologies for verification. Commonly, formal verification, such as model checking [15], [16], achieves full coverage of behavioral aspects and can ensure correctness for a DUV, but possibly suffers issues regarding scalability. In contrast, simulation-based methods involve more controllable run times, while input stimuli generation and concluding correctness depend on expert knowledge.

To deal with the challenges of the latter, various simulation-based methods have emerged with different advantages and disadvantages. Methods from the domain of SW, such as Fuzzing [17]–[20]; and Symbolic Execution [21], [22]; have shown to be feasible for the generation of input stimuli for DUVs. Such methods rely on guided input generation through assumptions and coverage-guided feedback information to handle larger input and state spaces. The technique of *Constrained Random Verification (CRV)* [23] enables a more effective constraining while being compatible with functional and code coverage aspects of the DUV. By internally utilizing SMT solvers, the constraints are turned into input stimuli systematically and automatically. As CRV for HW verification is well-established, tracking coverage metrics (functional and code) through commercial RTL simulators alongside the CRV methodology is possible. Various tailored variants have been proposed for verification beyond RTL, like CRAVE [24] for the SystemC modeling language, and for specific use-cases, tools such as riscv-dv [25] are suitable to explore CRV for the generation of RISC-V instructions.

B. Related Work

Since verification is essential yet time-consuming in chip development, hardware verification has accumulated extensive research on methodologies, techniques, and metrics to gauge its completeness. In this scope, HW verification is commonly split into formal verification methods [15], [26], aiming for thorough proofs of HW properties, and simulation-based methods [23], utilizing golden reference models and assertions to check for behavioral correctness. Within simulation-based methods, techniques like fuzzing, symbolic execution, machine learning techniques, and CRV are among the widely adopted methods. Particularly, the authors of [17], [18], [20] all describe recent state-of-the-art fuzzing-based techniques, utilizing fuzzers from the SW domain to generate input stimuli. Other works, such as [13], [27], explore the verification of peripherals through symbolic generation of input stimuli. Applying fuzzing or symbolic execution to processor and *Central Processing Unit (CPU)* designs offers additional constraints applicable to families of processors to improve the quality of results [11], [12], [22]. While fast, fuzzers require a lot of guidance and additional feedback (e.g., coverage-guided fuzzing), and when proposed as a general framework, require a lot of tailoring. Symbolic execution formally identifies inputs with symbolic variables that represent value sets, narrowing them as the behavioral description is explored. However, the thorough exploration of the input space is subject to state-space explosion, hence having possibly huge run times.

CRV provides a balanced input generation approach with constraints defining legal and illegal patterns and enables sequencing for complex circuits that would otherwise need extra guidance [23]. Various techniques explore extensions around CRV for modeling languages like SystemC [24], to improve the simulation run times [28], and to improve the quality of results [29]. The introduction of HW models at different levels of abstraction also proposed the need for methods covering across such levels of abstraction (e.g., system-level and RTL) [30], [31]. When utilizing VPs, analyzing and merging different models in *Transaction Level Modeling (TLM)* inside VPs and the RTL [13] or unifying coverage between SW and HW of the system [32] are particularly recent challenges [31]. Techniques to improve CRV, particularly relying on machine learning, such as Bayesian networks and Bayesian estimation [33]–[35], require knowledge about the parameters of the verification process. Although many works show the exploration through machine learning techniques, the effect of various parameters is often assumed or unknown, and a systematic exploration of verification parameters is neglected, although realistic [31]. As a result, understanding the role of verification parameters remains an open challenge in cross-level CRV-based verification.

In contrast to works that propose novel verification approaches or focus on the optimization for coverage closure, this work takes a complementary position by systematically characterizing the impact of verification parameters within an existing CRV-based cross-level verification approach. While other recent approaches leverage machine learning or coverage guidance to improve verification efficiency, they typically assume verification parameters to be fixed or implicitly tuned. Our work instead focuses on understanding how parameters such as instruction sequence length, mutation location, and mutation count influence the bug-finding effectiveness and the behavior of different coverage metrics.

III. Cross-Level Verification with CRV

The framework presented in this work performs verification for a given cross-level setup with CRV. It is utilized to explore the verification parameter space, spanned through different configurations of the length of input stimuli, the amount of mutations injected into the DUV, as well as the location of these mutations.

Figure 1 shows an overview of the proposed approach, with numbered steps guiding the flow. At ①, the figure shows the explored verification parameter space: the length of instruction

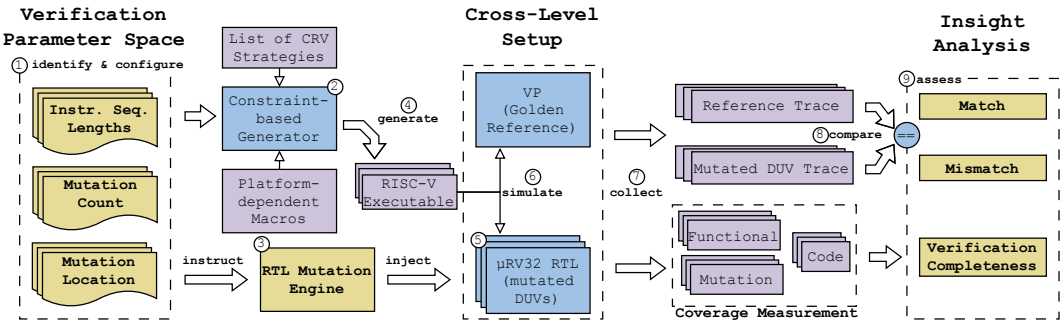


Figure 1: Overview of the proposed CRV-based Cross-Level Framework for Verification Parameter Space Exploration.

sequences, the number of mutations, and the location of mutations that are injected into the DUV. These parameters are used to configure and instruct the CRV tool (②), together with the different CRV strategies, and the RTL mutation engine (③). In particular, for this work, we explore the verification parameter spaces within the riscv-dv [25] framework together with a commercial RTL simulator. It should be noted that other tools and techniques to generate RISC-V instructions, such as Scala-based RISC-V Torture Test [36] or FORCE-RISCV [37], can be used instead of riscv-dv. A requirement for other tools is the availability of parameters (e.g., instruction sequence length) and a mechanism to execute instructions in the cross-level setup. The introduction of mutations is performed seperately from the instruction generation (③). The CRV tool generates instruction sequences that are compiled to platform specific executables (④) for the cross-level setup. In this setup, we utilize a RISC-V based *System-on-Chip (SoC)* for the RTL, namely MicroRV32 [38], with its corresponding RISC-V VP [39] platform. As the two platforms are binary compatible, the exact same executable is run on both models. Next, the RTL mutation engine injects one of the given mutations into one of the submodules of the DUV (⑤). The mutated RTL represents a DUV with a bug, hence varying the type and location of the mutation emulates a different fault. As part of this work, we provide the scripts to mutate any RTL design as open source to stimulate further research. The scripts utilize part of the capabilities to mutate modules provided by Yosys [40], with additional processing stages added, enabling a granular choice of submodules for the injection of mutations. The mutations flip bits, replace them with constants, or alter logic operations in expressions. We then filter out ineffective mutations or those whose effects are not observable at the outputs. Through the simulation (⑥) the VP and the DUV create execution traces, which are collected (⑦) for comparison (⑧). In this work, we utilize the predefined trace format from riscv-dv, consisting of the program counter, the instruction, the registers used and updated by the instruction, updated privileged registers by the instruction, the instruction binary, and the immediate operands. This cross-level execution avoids costly RTL-RTL co-simulation and allows longer instruction sequences to be executed efficiently, while mismatches between VP and RTL directly indicate faulty behavior. Additionally, coverage metrics, such as mutation coverage, RISC-V related functional coverage, and code coverage for RTL and VP are collected. Beyond traditional RTL-centric metrics, the VP enables enhancement and customization of the coverage measurement process. This gathers execution metrics such as value coverage and the unique sets of register and immediate values. The mutation coverage represents the amount of mutations killed (i.e., the framework detects a mismatch), the functional coverage represents the different functional aspects given in the RISC-V *Instruction Set Architecture (ISA)*, and the RTL code coverage represent metrics like statement coverage, mux and toggle coverage, branch coverage and more. The last step (⑨) assesses the outcome of the cross-level verification (i.e., which bugs led to faulty behavior) as well as the completeness of the verification. By integrating RTL-based and VP-based coverage metrics in a cross-level setup, the framework offers a holistic view of how verification parameters affect both coverage behavior and bug-finding effectiveness. Except for the final analysis, the parameter space exploration runs fully automated.

IV. Evaluation

A. Experimental Setup

For the evaluation, we explore the verification parameter space of our CRV-based cross-level verification with the following verification parameters:

P_{Ver1}: The length of instruction sequences is varied in the range 100, 1000 and 10 000. Resulting in three different executable lengths.

P_{Ver2}: To evaluate a reasonable number of faults, the number of mutations for each mutation target is fixed at 500, with an exception for the Fetch Unit due to its small size, where we inject 249 mutations. Resulting in 2749 $(5 \times 500 + 1 \times 249)$ distinctly mutated DUVs.

P_{Ver3}: Mutations are targeted to the full CPU as well as submodules ALU, Control Unit, Fetch Unit, Instruction Decoder and Multiply-Divide Unit, respectively. Resulting in six different mutation locations.

We apply the CRV strategies *Full Random*, *Arithmetic*, *Load-Store*, *Jump-Branch* and *Loop*, resulting in five different types of executables with different instruction corpora per CRV generation. Following the verification parameters above, we generate instruction sequences with each strategy, for each sequence length (P_{Ver1}) and apply them to each distinct DUV

Table I: Effect of generated instruction sequence length (P_{Ver1}) on mutation-based, functional and code coverage.

Instr. [#]	Mutation Coverage			Func. Cov. [%]	RTL Code Coverage		
	Killed [#]	Total [#]	Killed [%]		Branch Cov. [%]	Statement Cov. [%]	Toggle Cov. [%]
100	9663	13745	70.30	56.11	78.97	85.71	70.41
1000	9903	13745	72.05	57.61	78.84	85.53	70.64
10000	11254	13745	81.88	57.70	78.97	85.71	70.87

Table II: Breakdown of module-based mutations (P_{Ver3}) and the effect of number of generated instructions (P_{Ver1}) on mutation coverage, functional coverage and RTL code coverage.

Module	Instr. [#]	Mutation Coverage			Func. Cov. [%]	RTL Code Coverage		
		Killed [#]	Total [#]	Killed [%]		Branch Cov. [%]	Statement Cov. [%]	Toggle Cov. [%]
ALU	100	1963	2500	78.5	42.9	78.97	85.71	70.18
	1000	2086	2500	83.4	46.4	78.58	85.36	70.60
	10000	2217	2500	88.7	47.8	78.71	85.62	70.83
Control	100	1120	2500	44.8	39.9	78.20	85.18	70.07
	1000	1464	2500	58.6	44.7	78.71	85.45	70.45
	10000	2148	2500	85.9	44.5	78.97	85.71	70.77
Decode	100	1918	2500	76.7	42.0	78.46	85.27	70.14
	1000	1578	2500	63.1	47.1	78.58	85.36	70.51
	10000	1570	2500	62.8	47.3	78.20	85.18	70.16
Fetch	100	1188	1245	95.4	47.0	78.20	85.18	70.06
	1000	1190	1245	95.6	49.0	78.46	85.36	70.35
	10000	1206	1245	96.9	47.8	78.97	85.71	70.78
MulDiv	100	2116	2500	84.6	36.6	78.20	85.18	70.25
	1000	2226	2500	89.0	44.6	78.84	85.53	70.38
	10000	2378	2500	95.1	43.9	78.97	85.71	70.83
Core	100	1358	2500	54.3	43.6	78.20	85.18	70.17
	1000	1359	2500	54.4	46.1	78.58	85.36	70.47
	10000	1735	2500	69.4	47.3	78.97	85.71	70.85

(P_{Ver2} and P_{Ver3}). Therefore, DUVs with 500 mutations have 2500 $(5 \cdot 500)$ runs per instruction sequence length and 249 mutations result in 1245 $(5 \cdot 249)$ runs per instruction sequence length, respectively. Summing up each of these runs per instruction sequence length results in $13\,745$ runs $(5 \cdot 2500 + 1 \cdot 1245)$. In total, we performed $41\,235$ $(3 \cdot 13745)$ different cross-level verification runs, each resulting in coverage and mismatch measurements. The experiments were performed on a Intel(R) Xeon(R) Gold 6240 CPU with 36 cores @ $2.6\,\mathrm{GHz}$ with $376\,\mathrm{GB}$ memory running Ubuntu 22.04. Each run took around ten seconds of walltime on our setup. Simulations were performed through riscv-dv and a commercially available RTL simulation tool, while the accompanying SoC in RTL and the corresponding VP are available as open-source.

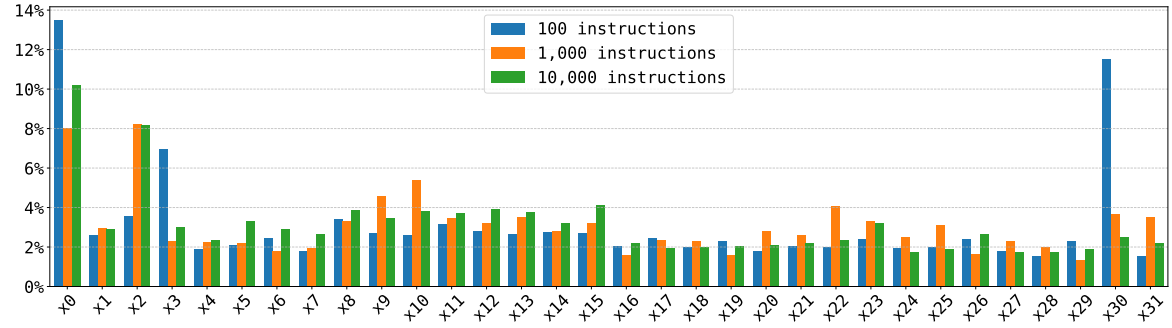
B. Evaluation Results

For the total $41\,235$ performed runs, we first break down the effectiveness and coverage metrics of the full design, in relation to P_{Ver1}. This allows us to observe this parameter in more isolation, when applied to the full CPU. After that, we illustrate the results of injecting the respective number of mutations (parameter P_{Ver2}) into each module separately, as described through the parameter P_{Ver3}. To assess result quality, we examine immediate value coverage in the instruction stream and analyze register file accesses, reporting total accesses and unique values per register.

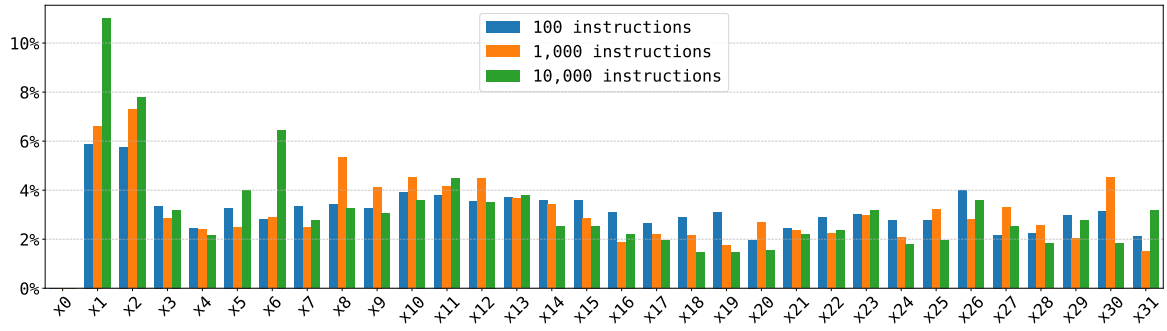
Table I provides an overview of detected mutations (i.e., mutation coverage), functional coverage and selected code coverage metrics (branch, statement, and toggle coverage) in relation to the verification parameter P_{Ver1}. The table is split into four parts: the leftmost column shows the parameter length of instruction for the CRV generation, second to fourth columns show the mutation coverage with the absolute number of detected mutations in the second column, and the relative amount in the fourth column, the functional coverage given by riscv-dv in the fifth column, and the code coverage obtained from the RTL simulation in the sixth to eighth columns. For code coverage, branch, statement, and toggle coverage are displayed as representatives, as they feature the largest number of bins for the RTL core (over 1000 bins compared to 161 bins or less for the other code coverage metrics), providing more information on different parts of the RTL code.

Increasing the instruction count per verification run leads to more mutations being detected through DUV-VP mismatches. Even for only 100 generated instructions, already over $70\,\%$ of the $13\,745$ injected mutations are killed by CRV. This number is increased to almost $82\,\%$ for $10\,000$ instructions. Functional coverage ranges from $56.11\,\%$ to $57.70\,\%$, showing only marginal increase with increasing instruction sequence length. Generally, code coverage values are much higher than functional coverage, but once again behave very similarly between the number of instructions. Values range from $70.41\,\%$ for toggle coverage of 100 instructions to $85.71\,\%$ for statement coverage of $10\,000$ instructions. Note that longer instruction sequences are not generated on the corpus of shorter sequences but generated anew, hence explaining the slight drop in branch and statement coverage between 100 and 1000 instructions. Despite this fact, the results provide an indication on the effect of P_{Ver1} due to the high number of $13\,745$ runs per instruction length.

Table II breaks down the information summarized in Table I for each module (i.e., unfolding verification parameters P_{Ver1} and P_{Ver3} in one table). The table is split into five parts: the leftmost column shows each module from verification parameter P_{Ver3} and the number of mutations applied to



(a) Relative number of register accesses (%)



(b) Relative number of unique values per register (%)

Figure 2: Relative amount of generated instructions versus number of register accesses and unique values per register.

it. The second column follows with the instruction sequence length that was used for each module, while the third to fifth columns show the mutation coverage, the sixth column shows the functional coverage given by riscv-dv and the seventh to ninth columns show the RTL code coverage obtained for the particular module. A change in coverage behavior is noted for different mutation locations, with a notable difference (34.1 percentage points), for example, between the Decode unit (62.8 %) and the Fetch unit (96.9 %) for an instruction sequence length of 10 000. Across different instruction sequence lengths, the effects stay mostly consistent with the behavior observed previously in Table I, with the exception of the Decode unit. Changes in functional coverage generally show a slight increase with an increasing instruction sequence length, where the starkest difference is observed for the MulDiv unit, with 36.6 % for 100 instructions and 43.9 % for 10 000 instructions. Regarding code coverage, values behave very similar across verification parameters P_{Ver1} and P_{Ver3}.

Table III shows the number of total immediate values and unique immediate values for the six different types of immediate fields found in RISC-V instructions. The leftmost column shows the immediate field together with its number of bits in parentheses (e.g., I-Imm being the 12 bit immediate for the I-type instructions), the second column breaks down the values for the different instruction sequence lengths (P_{Ver1}), the third column shows the total number of immediates of that type present, while the fourth and fifth columns present the absolute and relative number of uniquely generated immediate values. The Shamt field (for shift amounts) is covered completely, and

Table III: Number of immediate values and unique immediate values for different instruction types and fields for each instruction sequence length (P_{Ver1}).

Imm. Type (Bits)	Instr. [#]	Total Values [#]	Unique Values [#]	Value Coverage [%]
Shamt (5)	100	1399	32	100%
	1000	9680	32	100%
	10000	36774	32	100%
I-Imm (12)	100	4853	1117	27.27%
	1000	41242	2250	54.93%
	10000	118662	4060	99.12%
S-Imm (12)	100	4221	268	6.54%
	1000	14087	976	23.83%
	10000	32966	3303	80.64%
B-Imm (13)	100	743	63	1.54%
	1000	4862	131	3.20%
	10000	16667	180	4.39%
J-Imm (20)	100	2143	96	0.01%
	1000	3698	102	0.01%
	10000	16088	93	0.01%
U-Imm (20)	100	3324	443	0.04%
	1000	8687	543	0.05%
	10000	23139	988	0.09%

the I-Imm field becomes covered almost completely (99.12 %) when generating 10 000 instructions, while other fields show lower values. The S-Imm field is covered to around 81 % after 10 000 instructions, with the remaining fields (B-Imm, J-Imm and U-Imm) staying below 5 %. The remaining fields (B-Imm, J-Imm, U-Imm) show only a slight rise with more instructions, underscoring the exponential growth of state space for each added immediate bit. The I-Imm and S-Imm fields on the other hand exhibit a much larger increase depending on instruction sequence length, starting from 27.27 % and 6.54 % respectively for 100 instructions.

Figure 2 shows two plots illustrating the different register file accesses (Figure 2a) and unique values (Figure 2b) handled per register respectively. Both plots map relative register ac-

cesses and unique values for each instruction sequence length. Note that, as x0 is a special purpose register with the constant zero value, it will always only hold one unique value. For 100 instructions, the highest percentage of register accesses for x0 is around 14 %, and for 10 000 instructions, this number decreases to 10 %, showing lower extremes for longer instruction sequences (in this regard, also notice register x30). For 1000 instructions, x2 and x0 manifest as outliers, with both being responsible for around 8 % of register accesses. As for unique values, the opposite effect can be observed. Here, the highest relative number for 100 instructions is around 6 % of unique values for register x1, while for 1000 instructions, roughly 7 % of unique values are stored in register x2 and for 10 000 instructions, 11 % of unique values are stored in x1.

C. Discussion of Exploration Results

Below we discuss our exploration results and the impact of the three verification parameters. We examine only relative differences across parameters and not the coverage closure to gain insight, rather than to optimize them. Also note that, while the numbers presented are specific to our CRV setup, we expect that the use of other tools that can be plugged into our framework, as mentioned above, would reveal similar trends.

First, we consider P_{Ver1} across all evaluations. Intuitively, the largest effect may be expected from this parameter, as it increases the chance of constrained random instruction generation to hit certain coverage targets. Looking at Table I, increasing the number of instructions does indeed increase mutation coverage, but the effect is limited, only furthering coverage closure by 39 %, despite a 100 times increase in instruction sequence length. For functional and code coverage, this effect is decreased even further. However, our functional coverage measure contains very detailed targets (e.g., bit-wise opposite values for logical operations), as well as coverage of *Control and Status Registers (CSRs)*, which are not covered by the employed CRV strategies. Hence, a direct rise in this metric is not guaranteed. Mutation coverage illustrates this, as it exists between RTL code coverage (offering limited interpretability) and behavior-tracking functional coverage. Regarding the coverage of immediate values shown in Table III, increasing instruction sequence length has a much larger effect, where almost complete coverage closure can be achieved for the I-Imm value by increasing from 100 to 10 000 instructions. A similar effect can be observed for the S-Imm value. For the much larger coverage spaces of J-Imm and U-Imm values, the increased coverage does not show directly, which may be attributed to the large coverage space and the much reduced number of instructions using these immediate encodings.

The generated instruction sequences use registers fairly uniformly regarding access frequency and unique values. While for the number of register accesses, less outliers occur in the relative distribution for 10 000 instructions, the distribution of unique values does not seem to be more evenly distributed for increased instruction sequence lengths. A possible explanation for this might be the fact that more unique values are generated for 10 000 instructions, but register distribution for generated instructions may not be as randomized.

As for P_{Ver2}, we can observe the difference between the Fetch unit with 249 mutations and remaining modules with 500 mutations in Table II. Here, fewer mutations raise mutation coverage but barely affect functional or code coverage.

Finally, looking at P_{Ver3}, Table II shows that the location of mutation injection can make a significant difference regarding mutation coverage. While this effect is slightly reduced between instruction sequence lengths (e.g., 50.6 percentage points between Control unit and Fetch unit for 100 instructions and 34.1 percentage points between Decode Unit and Fetch unit for 10 000 instructions), a divergence can still be observed. Regarding mutations injected at arbitrary locations (row *Core* in Table II), these are generally harder to catch, due to the increased spread of mutations. The Fetch unit shows the highest mutation coverage at all sequence lengths, likely because it is less complex than other modules. However, these findings do not translate to functional or code coverage. While for the former, there are at least some minor differences between mutation locations (around 10 percentage points), for the latter, differences between modules are marginal, indicating that these coverage values converge in a similar fashion.

In summary, our systematic exploration provides the following insights on the effect of different verification parameters on verification quality:

- Increased instruction count achieves higher mutation coverage and cover a higher range of immediate values.
- Higher instruction count does not imply more functional or code coverage.
- The use of functional and code coverage to gauge verification quality needs further investigation, considering the limited correlation with mutation coverage in our evaluation.
- Mutations targeted at specific locations of a design may be killed with higher precision than random mutation locations.
- This divergence between locations may be a useful indicator of which parts of the design might require further attention beyond CRV during the verification process.

V. Conclusion

In this work, we proposed a combined approach to explore three verification parameters for CRV in a cross-level setup between RTL HW and system-level VPs. Through the systematic exploration of these three verification parameters, the varying effect of these parameters on different coverage metrics has been demonstrated. Particularly, for mutation coverage, functional coverage and RTL code coverage, dependencies on the generated instruction sequence length and the location of the injected mutation were shown. By making the tool for injecting the mutations with module granularity into the designs available as open source, we aim to stimulate further research. To boost CRV-based techniques further, in the future, we plan to explore feedback-based methods to adapt CRV strategies, and parameters to guide input generation towards undetected mutations/bugs.

REFERENCES

[1] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[2] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[3] A. Gerstlauer, C. Haubelt, A. D. Pimentel *et al.*, "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, 2009.

[4] B. Bailey, F. Balarin, and M. McNamara, *Tlm-driven design and verification methodology*. Cadence Design Systems, 2010.

[5] "Ieee standard for standard systemc® language reference manual - corrigendum 1," *IEEE Std 1666-2023/Cor 1-2025 (Corrigendum to IEEE Std 1666-2023)*, pp. 1–22, 2025.

[6] C. Hazott, F. Stögmüller, and D. Große, "Using virtual prototypes and metamorphic testing to verify the hardware/software-stack of embedded graphics libraries," *Integration*, vol. 101, p. 102320, 2025.

[7] S. Tempel, V. Herdt, and R. Drechsler, "Symex-vp: an open source virtual prototype for os-agnostic concolic testing of iot firmware," *Journal of Systems Architecture*, vol. 126, p. 102456, 2022.

[8] B. Lin and F. Xie, "A systematic investigation of state-of-the-art systemc verification," *Journal of Circuits, Systems and Computers*, vol. 29, no. 15, p. 2030013, 2020.

[9] J.-H. Oetjens, N. Bannow, M. Becker *et al.*, "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *Proceedings of the 51st annual design automation conference*, 2014, pp. 1–6.

[10] P. Pieper, V. Herdt, and R. Drechsler, "Advanced environment modeling and interaction in an open source risc-v virtual prototype," in *Proceedings of the Great Lakes Symposium on VLSI 2022*, 2022, pp. 193–197.

[11] N. Kabylkas, T. Thorn, S. Srinath *et al.*, "Effective processor verification with logic fuzzer enhanced co-simulation," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 667–678.

[12] N. Bruns, V. Herdt, D. Große *et al.*, "Efficient cross-level processor verification using coverage-guided fuzzing," in *Proceedings of the Great Lakes Symposium on VLSI 2022*, 2022, pp. 97–103.

[13] K. A. Rudkowski, S. Ahmadi-Pour, and R. Drechsler, "Crossym: Cross-level verification of systemc peripherals using symbolic execution," in *2025 IEEE 28th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2025, pp. 153–156.

[14] R. Pelánek, "Fighting state space explosion: Review and evaluation," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2008, pp. 37–52.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled, "Model checking the mit press," *Cambridge, Massachusetts, London, UK*, vol. 988, 1999.

[16] R. Drechsler, *Advanced formal verification*. Springer, 2004.

[17] T. Trippel, K. G. Shin, A. Chernyakhovsky *et al.*, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3237–3254.

[18] K. Laeufer, J. Koenig, D. Kim *et al.*, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[19] Y. Cheng, H. Zou, J. He *et al.*, "Mmfuzz: Towards enhancing rtl fuzz testing using metric feedbacks based on markov chain," in *2023 IEEE 32nd Asian Test Symposium (ATS)*. IEEE, 2023, pp. 1–6.

[20] J. Hur, S. Song, D. Kwon *et al.*, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1286–1303.

[21] K. Ryan and C. Sturton, "Sylvia: Countering the path explosion problem in the symbolic execution of hardware designs," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2023, pp. 110–121.

[22] N. Bruns, V. Herdt, and R. Drechsler, "Processor verification using symbolic execution: A risc-v case-study," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[23] A. B. Mehta, "Asic/soc functional design verification," *A Comprehensive Guide To Technologies and Methodologies*, 2018.

[24] F. Haedicke, H. M. Le, D. Große *et al.*, "Crave: An advanced constrained random verification environment for systemc," in *2012 International Symposium on System on Chip (SoC)*. IEEE, 2012, pp. 1–7.

[25] CHIPS Alliance, "RISCV-DV." [Online]. Available: https://github.com/chipsalliance/riscv-dv

[26] T. Kropf, *Introduction to formal hardware verification*. Springer Science & Business Media, 2013.

[27] P. Pieper, V. Herdt, D. Große *et al.*, "Verifying systemc tlm peripherals using modern c++ symbolic execution tools," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1177–1182.

[28] S. M. Ambalakkat and E. G. Nelson, "Simulation runtime optimization of constrained random verification using machine learning algorithms," in *Design and Verification Conf.(DVCON)*, 2019.

[29] A. Nazi, Q. Huang, H. Shojaei *et al.*, "Adaptive test generation for fast functional coverage closure," *DVCON USA*, 2022.

[30] S. Vinco, N. Bombieri, D. J. Pagliari *et al.*, "A cross-level verification methodology for digital ips augmented with embedded timing monitors," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 3, pp. 1–23, 2019.

[31] A. Mahmoudi, A. Nešković, C. Thermann *et al.*, "A systematic mapping study on systemc/tlm modeling capabilities in new research domains," *ACM Transactions on Design Automation of Electronic Systems*.

[32] C. Hazott and D. Große, "Relation coverage: A new paradigm for hardware/software testing," in *2024 IEEE European Test Symposium (ETS)*. IEEE, 2024, pp. 1–4.

[33] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 286–291.

[34] Q. Huang, H. Shojaei, F. Zyda *et al.*, "Test parameter tuning with blackbox optimization: A simple yet effective way to improve coverage," in *Proceedings of the design and verification conference and exhibition US (DVCon)*, 2022.

[35] B. Kumar, G. Parthasarathy, S. Nanda *et al.*, "Optimizing constrained random verification with ml and bayesian estimation," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.

[36] Y. Lee and H. Cook, "RISC-V Torture Test." [Online]. Available: https://github.com/ucb-bar/riscv-torture

[37] OpenHW Group, "FORCE-RISCV." [Online]. Available: https://github.com/openhwgroup/force-riscv

[38] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "The microrv32 framework: An accessible and configurable open source risc-v cross-level platform for education and research," *Journal of Systems Architecture*, vol. 133, p. 102757, 2022.

[39] V. Herdt, D. Große, P. Pieper *et al.*, "Risc-v based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 2020.

[40] D. Shah, E. Hung, C. Wolf *et al.*, "Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 1–4.