CrosSym: Cross-Level Verification of SystemC Peripherals using Symbolic Execution

Karl Aaron Rudkowski[1] Sallar Ahmadi-Pour[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{karlaaron,sallar,drechsler}@uni-bremen.de

***Abstract*—Modern hardware design is challenged by the ever-increasing complexity of designs. An important step is hardware verification, where Virtual Prototypes (VPs) can be used for a cross-level verification throughout the refinements. Peripherals are relevant targets, because they characteristically implement a wide range of integral tasks. Symbolic execution is a popular verification method, but has been applied to SystemC peripherals only once, and never for a cross-level verification. We propose CrosSym, the first method to verify peripherals at both Register Transfer Level (RTL) and Transaction Level Modelling (TLM) abstraction with symbolic execution, explicitly supporting cross-level. Our extensive evaluation explores (1) the performance costs of two abstraction levels, (2) our approach's suitability for a full verification, using three peripherals, (3) it's bug finding capabilities by killing over 1500 mutants in under $15\,\text{min}$. In the latter, most scenarios found 97+% of the observable mutations.**

***Index Terms*—Virtual Prototypes, Symbolic Execution, RTL, TLM, Cross-Level**

I. Introduction

In modern hardware design, VPs take a central role. A VP is an executable software model of hardware, which is commonly implemented in SystemC [1]. With a VP, software design and verification can start early on, long before any real hardware exists. Verification, against both a specification and higher abstraction level models, is an important step to make bug fixes cheaper and easier. One popular approach for this is symbolic execution, which evaluates a device using symbolic inputs, each representing sets of concrete inputs. Thus, it is not restricted to concrete test cases, and can offer a more complete reasoning. One important verification target are the peripherals. They implement a wide range of tasks, e.g. communication protocols, or modern accelerators for neural networks [2]. Because of these characteristic differences to the processor, they require separate attention. However, modern symbolic execution tools cannot be applied to SystemC devices, mainly because the kernel is implemented using system threads. To benefit from them anyway, current state of the art replaces the kernel with alternative implementations [3], [4]. These are limited to high-level features, and inherently not capable of a cross-level verification for lower levels like RTL. To address this gap of cross-level verification methodologies, we propose CrosSym, the first approach of this kind. We contribute:

This work was supported by the German Federal Ministry of Education and Research (BMBF) under grant no. 16ME0127 and no. 16ME0135 (Scale4Edge), and no. 01IW22002 (ECXL) .

1) A lightweight SystemC replacement kernel, which supports the relevant features of both RTL and TLM. It is the first SystemC kernel enabling cross-level symbolic execution. An additional benefit is it's reduced complexity, which enables a faster verification.
2) A workflow offering three possible verification scenarios, namely standalone RTL or TLM verification, and a cross-level comparison between the two.
3) An examination of (1) the performance costs of supporting RTL *and* TLM, by comparing to the state of the art [4], and our approach's effectiveness (2) for a full verification, using three peripherals, (3) for bug finding, by killing over 1500 mutants in under $15\,\text{min}$.

Related Work: Prior work spans across a wide range of techniques. Model checking [5]–[8] formally proves properties for a design. However, these approaches often require intermediate representations, which limit their applicability. Approaches originating from the software domain can mitigate this issue. Coverage-guided fuzzing [9], [10] and concolic testing [11]–[13] are limited to the generated concrete test cases. In contrast, symbolic execution offers the possibility of a complete exploration by replacing concrete values with symbolic variables. At the RTL, C++ models, e.g. generated by Verilator [14], are used [15]–[17]. These do not support TLM, and thus cross-level verification. For higher abstractions in SystemC, the state of the art are custom SystemC schedulers [3], [4]. Their supported features are limited, and RTL or cross-level verification is impossible. Only SymSysC [4] explicitly targets peripherals. While it is thus closest to our goal of a cross-level peripheral verification using symbolic execution, TLM support is limited and RTL missing. To the best of our knowledge, there are no other approaches which achieve our goal, which makes CrosSym the first of it's kind.

II. Preliminaries

SystemC is a modelling language that encapsulates hardware components in modules. Each module's functionality is realised with methods or threads. These processes register themselves for execution on events such as clock edges, wait times, or incoming inputs. Inter-module communication can be based on signals and ports, or, at a higher abstraction level, TLM sockets.

Symbolic Execution is a verification technique from the software domain. There exist several software-focused engines

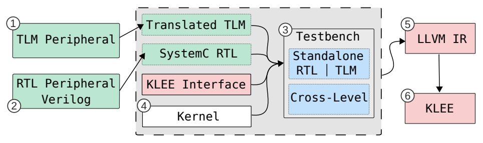

Fig. 1. Overview Verification Workflow.

like KLEE [18] or Angr [19]. The idea is to represent sets of concrete values with symbolic variables. During the execution, instructions involving these symbolic variables are evaluated using a Satisfiability Modulo Theory (SMT) solver. This can result in forking the execution (branch instructions), or generating concrete inputs that trigger errors. Thereby, the program can be checked for generic failures (e.g. divide by zero), and assertion violations.

III. Cross-Level Verification Using a Replacement Kernel

When verifying SystemC peripherals, the implementation of the original SystemC kernel causes two main obstacles.

1) The `SC_THREADS`, commonly used in TLM models, are realised using system threads. However, threading is currently not supported by the state-of-the-art symbolic execution tools due to the large overhead in verifying multithreaded software.
2) The signal-port mechanism, commonly used in RTL models, uses `mutexes`, which are likewise not supported.

These make an alternative kernel implementation, designed for symbolic execution, a solution worth considering. Such a replacement kernel was proposed for example by Pieper et al [4], who focused solely on TLM peripherals. Noteworthy is the thread translation, which addresses the first problem. However, the kernel was limited to TLM features, and misses all RTL features. As a consequence, the authors do not address the aforementioned second problem. Thus, while the idea itself is promising and proved effective in verifying a TLM Platform Level Interrupt Controller (PLIC), the verification of RTL peripherals is not possible and poses unique challenges to consider. In the following sections, CrosSym is presented, which pursues a similar idea to support both TLM *and* RTL peripherals.

Verification Workflow: In Figure 1, we give an overview of the proposed verification workflow. The desired Design under Verification (DUV) can be provided at both the RTL and TLM level. The TLM description ① is usually readily available in SystemC, therefore, only the threads need to be translated as described in [4]. Similarly, the RTL description ② can be implemented in SystemC, possibly with a transcompilation using Verilator [14]. Such open source tools, and SystemC's RTL support, make this extra step worth considering. In the test bench ③, assertions define the desired behaviour, and KLEE interface methods declare symbolic variables. The tests can realise three possible verification scenarios:

(a) Standalone verification of a TLM device
(b) Standalone verification of a RTL device
(c) Cross-level verification RTL⇔TLM

If both RTL and TLM implementations are available, the test bench can cover all three scenarios in separate tests. The translated device, optimised kernel ④, and test bench are compiled into the LLVM *Intermediate Representation*, using the Clang C++ compiler ⑤. Finally, the peripheral can be verified using KLEE ⑥.

Application-Specific Optimisation: The original SystemC kernel supports many concepts for a wide range of applications. However, this complexity can negatively impact the verification time, which is already a concern in comparison to testing approaches such as fuzzing. For our desired goal of verifying peripherals in isolation, the set of necessary features is smaller. Including unused features would disadvantage the symbolic execution, possibly to the point of rendering it ineffective. As a consequence, we carefully chose the supported features to cover our target use case. An example of such a concept is asynchronous waiting, which, amongst other things, introduces `mutexes` into the signal-port mechanism. This is not necessary, since we do not aim to verify entire systems or interactions between multiple peripherals.

Replacement Kernel: Our kernel supports the TLM concepts of SymSysC with improvements in the event notification system. However, it's main benefit are the RTL concepts of static sensitivity, the clock, and signals and ports with delta cycles. These make the verification of RTL peripherals possible. Table I shows a comparison of the supported features between SymSysC and our kernel. The first column lists all the different features of SymSysC and our work. The middle column shows the ones featured in our approach, while the right one shows those of SymSysC [4].

TABLE I
Feature comparison between SymSysC and our kernel.

	Our Work	**SymSysC [4]**
Time	✓	✓
`wait(time)`	✓	✓
Threads & Methods	✓	✓
Static sensitivity	✓	×
Events	✓	(✓)
`wait(event)`	✓	(✓)
Signals & Ports	✓	×
Delta cycles	✓	×
Clock	✓	×

IV. Evaluation

In order to evaluate CrosSym, we concerned ourselves with three research questions:

RQ1 How does supporting the verification at both RTL and TLM, as opposed to only TLM, influence the performance?
RQ2 Can our approach fully explore different peripherals?
RQ3 Can our approach offer effective bug finding?

Our experiments are based on the verification of three peripherals available at RTL and TLM: a RISC-V PLIC, as well as modules to compute the Greatest Common Divisor

TABLE II
PERFORMANCE COMPARISON.

	SymSysC [4]				**This Paper**				**% Increase**	
	Paths compl.	partial	**Time**	**Mem**	**Paths** compl.	partial	**Time**	**Mem**	**Time**	**Mem**
1	63	1	52000.40	199.56	63	1	73386.36	212.21	41.13	6.34
2	189	0	68140.00	331.00	189	0	72373.11	360.18	6.21	8.82
3	91	49	Timeout	198.13	32	34	Timeout	155.62	-	-21.46
4	970	198	23539.59	117.08	970	198	48871.87	143.14	107.62	22.26
5	54	386	Timeout	183.31	5	386	Timeout	143.54	-	-21.7

TABLE III
PERIPHERAL VERIFICATION. E1-4=ERROR1-4. L1-2=LIMIT1-2.

Test			**Paths** compl.	partial	**Time**	**Solver**	**Memory**	**Comment**
1	PLIC	T1	168	0	271.29	86.11%	211.75	✓
2		T2	0	473	487.95	91.65%	334.41	E1
3		T3	2013	4860	86401.01	97.55%	2887.92	L1
4		EQ2	29	203	Timeout	98.44%	266.58	E2
5		EQ3	29	122	Timeout	98.63%	271.50	E1
6	GCD	RTL	19	110	Timeout	99.78%	144.68	E3, L2
7		TLM	54	2258	Timeout	99.82%	128.05	E4, L2
8		EQ	48	446	Timeout	99.64%	178.80	E3, E4 L2
9	Map	RTL	18	0	30624.99	98.81%	202.26	✓
10		TLM	1	0	0.25	5.51%	114.22	✓
11		EQ	18	0	30856.59	98.80%	206.45	✓

(GCD) and value mappings for a list. We used KLEE [18] (version 2.3) with the STP [20] solver. One execution runs a maximum of 24 h, and can use at most 4000 MiB of working memory. A single solver query can take at most 120 s. The search strategy is Breadth First Search (BFS).

Performance Comparison: Since our replacement kernel supports not only TLM, but also RTL peripherals, the added complexity can worsen performance. This is important because the verification should find results with reasonable resource usage. In order to evaluate the impact of our additional features, we repeat the case study from the SymSysC paper [4], once with the SymSysC kernel, and once with ours. The case study featured a RISC-V TLM PLIC, specifically the FE310 configuration based on the SiFive FE310 SoC [21]. It manages global interrupts, and notifies the targets. The interrupt order is determined by their individual priorities, as well as the per-target priority threshold. The first three tests consider (1) a basic case of triggering one symbolic interrupt number, (2) additionally with a symbolic priority and threshold, (3) two symbolic interrupts, each with a symbolic priority. The last two cases concern read/write transactions over the TLM socket.

Table II shows the run metrics for both versions. It is split into four parts, first defining the test case, then giving the performance of SymSysC [4] and our approach, and finally summarising the increased resource usage. In the middle two parts, the same columns list the complete and partially executed paths, and the time (in s) and average memory (in MiB) used by KLEE. In the final part, the increase in time and memory of our approach is given in percent. Both versions found the same errors, which are not listed for brevity. Our kernel comes with a penalty in both time and memory usage. Tests #3 and #5 are special cases, as they time out regardless of the kernel. Here, our *decreased* memory usage simply comes from the slower execution. Our method did not get as far and explored fewer paths, meaning fewer opportunities for memory-intense queries. While this shortcoming is of course not ideal, the detected errors remained the same. Furthermore, the aggravated resource use supports choosing a replacement kernel. By implementing only necessary features, we minimised the overhead. Without them and their associated costs, RTL peripherals cannot be verified. However, adding others would worsen the performance without advantages to our use case. Regarding the benefits of our kernel's features, the following discusses the verification of RTL peripherals.

Peripheral Device Verification: We considered all three peripherals. The first three RTL PLIC test cases mirror tests #1-#3 from Table II. Tests #4 & #5 mirror tests #2 & #3, but applied to the cross-level. For the new peripherals, there are three test cases each: RTL standalone, TLM standalone, and cross-level. The GCD module calculates the greatest common divisor of two input values. For the tests, these two inputs are made symbolic. The Map module maps up to eight input values to one output value each by applying a user-defined mapping value. Of these eight inputs, only two are made symbolic, in addition to the mapping value. The output values are independent from each other, and thus a faster verification is possible. Table III shows the results for each test case. First, statistics related to the code are given in the form of the number of explored complete/partial paths and the total executed instructions. Following, we list the run's complete duration (in s), the percentage of time spent in the SMT solver, and the average memory usage (MiB). Finally, the result of the test case is summarised. For the RTL PLIC, the relationship between the priority and the threshold was inverse (**E1**), and it handles priority values greater than the user-defined maximum differently (**E2**). While it's known implementation errors were identified, both memory (**L1**) and time limit (**Timeout** after 24 h) were encountered. The other DUV with a functional error was the GCD module. At TLM, an internal conversion can lead to wrong results (**E4**), and an infinite loop is possible (**E3**, or **Timeout** in #7). Furthermore, (solver) timeouts occur (**L2,Timeout**). This originates from the significantly larger state space, effected by a symbolic loop condition. Finally, the Map module was the only one reliably finishing without any timeouts. It is comparatively simple in terms of calculations and branch conditions.

In conclusion, we evaluated three different peripherals, found bugs and extensively explored the DUVs in a reasonable time frame. Future research has to identify solutions for large state spaces (see tests #6 to #8), and time-consuming complex paths (see tests #4 and #5). To better understand our approach's bug-finding capabilities, the following section deals with targeting mutations inserted into the DUVs.

Effective Mutation-Based Bug Finding: Our approach usually found existing errors early on. This is an important verification goal to minimise the danger of the unexplored peripheral code. In order to evaluate how suited our approach is for finding existing errors, we inserted mutations into the three DUVs using an automated script. For each mutant, the

TABLE IV
MUTATION TEST RESULTS. O[0-5]=OBSERVATION[0-5].

DUV		Mutants	Tests		Alive	Killed	%	Comment
PLIC	RTL	1067	T	3201	1052	2149	67.14	O1
			EQ	2134	2132	2	00.09	O4
	TLM	212	T	636	196	440	69.18	O1
			EQ	424	303	121	28.54	O4
GCD	RTL	87	T	87	1	86	99.28	O5
			EQ	87	0	87	100	✓
	TLM	19	T	19	0	19	100	✓
			EQ	19	12	7	36.84	O2
Map	RTL	85	T	85	0	85	100	✓
			EQ	85	0	85	100	✓
	TLM	35	T	35	0	35	100	✓
			EQ	35	1	34	97.14	O5

test cases discussed above were executed. We chose over 1500 that we observed to change the peripherals' behaviour. Table IV shows the results. First, the mutated DUV as well as the amount of observable mutants and total executed test cases are given. These latter are divided into standalone (T) and cross-level (EQ). For both, we divide the total amount into those that did not find an error and those that did. Following, we list the percentage of test cases that killed the mutant. Finally, the result is summarised. Some test cases timed out without detecting an error, even though the same mutant was killed by another test. While the short runtime can be responsible (**O5**), patterns can usually be observed. First, the PLIC functionality is split into multiple tests. Consequently, not all of these cover the mutated behaviour (**O1**). Additionally, the PLIC's cross-level tests are slower (see Table III), which is detrimental with the very short run time of 15 min. Second, the GCD module highlights the structural differences of RTL and TLM regarding the state tree. In TLM standalone, mutants can be killed in 100 queries or fewer. However, in the cross-level, with the intact RTL tree, thousands of queries might not suffice to venture deep enough into the tree. This supports the need for research into the search strategies. Here, neither BFS nor KLEE's default offer promising results.

In the mutation-based evaluation, our approach found over 1500 mutants fast ($<$15 min). 7 out of 12 scenarios found $\geq$97% of their respective mutations. Lastly, we identified the open question of an appropriate search strategy for symbolic execution of hardware models.

V. CONCLUSIONS

In this paper we presented CrosSym, the first cross-level verification method for peripherals using symbolic execution. Due to the obstacles to symbolic execution that originate from the original SystemC implementation, we propose a replacement kernel. This kernel supports key features of both TLM and RTL modelling, while being optimised for symbolic execution of peripherals in isolation. In a comparison with the state of the art, our approach maintained effectiveness at the TLM level. Further, with our novel set of features, we evaluated the effectiveness of verifying TLM and RTL peripherals, in a standalone as well as in a cross-level scenario. Additionally, our approach identified over 1500 mutations within 15 minutes. In most scenarios, we found over 97% of the respective device's overall observable mutations in this timeframe. For future work, we plan to identify more appropriate search heuristics for symbolically executing peripherals, and thus increase the manageable device complexity.

REFERENCES

[1] "IEEE Standard for Standard SystemC® Language Reference Manual," *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, pp. 1–618, 2023.

[2] J. L. Hennessy *et al.*, "A New Golden Age for Computer Architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[3] B. Lin *et al.*, "Generating High Coverage Tests for SystemC Designs Using Symbolic Execution," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 166–171.

[4] P. Pieper *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.

[5] P. Herber *et al.*, "STATE – A SystemC to Timed Automata Transformation Engine," in *2015 IEEE 17th ICHPCC, 2015 IEEE 7th CSS, and 2015 IEEE 12th ICESS*, 2015, pp. 1074–1077.

[6] H. M. Le *et al.*, "Towards Formal Verification of Real-World SystemC TLM Peripheral Models - A Case Study," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 1160–1163.

[7] S. Deng *et al.*, "Bounded Model Checking for RTL Circuits Based on Algorithm Abstraction Refinement," in *2006 8th International Conference on Solid-State and Integrated Circuit Technology Proceedings*, 2006, pp. 2082–2084.

[8] P. Bavonparadon *et al.*, "RTL Formal Verification of Embedded Processors," in *2002 IEEE International Conference on Industrial Technology, 2002. IEEE ICIT '02.*, vol. 1, 2002, pp. 667–672.

[9] N. Bruns *et al.*, "Efficient Cross-Level Processor Verification Using Coverage-Guided Fuzzing," in *Proceedings of the Great Lakes Symposium on VLSI 2022*, ser. GLSVLSI '22, 2022, p. 97–103.

[10] S. Ahmadi-Pour *et al.*, "Synergistic Verification of Hardware Peripherals through Virtual Prototype Aided Cross-Level Methodology Leveraging Coverage-Guided Fuzzing and Co-Simulation," *Chips*, vol. 2, pp. 195–208, 2023.

[11] A. Ahmed *et al.*, "Directed Test Generation Using Concolic Testing on RTL Models," in *2018 DATE*, 2018, pp. 1538–1543.

[12] Y. Lyu *et al.*, "Scalable Concolic Testing of RTL Models," *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 979–991, 2021.

[13] B. Lin *et al.*, "Concolic Testing of SystemC Designs," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, 2018, pp. 1–7.

[14] (2004) Verilator Compiler. [Online]. Available: https://www.veripool.org/verilator/

[15] Y. Zhang *et al.*, "Automatic Generation of High-Coverage Tests for RTL Designs Using Software Techniques and Tools," in *2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA)*, 2016, pp. 856–861.

[16] N. Bruns *et al.*, "Processor Verification using Symbolic Execution: A RISC-V Case-Study," in *2023 DATE*, 2023, pp. 1–6.

[17] R. Zhang *et al.*, "End-to-End Automated Exploit Generation for Validating the Security of Processor Designs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 815–827.

[18] C. Cadar *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, p. 209–224.

[19] Y. Shoshitaishvili *et al.*, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[20] V. Ganesh *et al.*, "A Decision Procedure for Bit-Vectors and Arrays," in *Computer Aided Verification*, W. Damm *et al.*, Eds., 2007, pp. 519–531.

[21] (2020) SiFive FE310-G000 Manual. [Online]. Available: https://static.dev.sifive.com/FE310-G000.pdf