Relation Coverage: A New Paradigm for Hardware/Software Testing

Christoph Hazott Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
christoph.hazott@jku.at daniel.grosse@jku.at

***Abstract*—While the *Hardware* (HW) domain and the *Software* (SW) domain use the concept of coverage to measure the thoroughness of tests, there isn't an established common metric that applies to both worlds. In this paper we make two major contributions: First, leveraging the abstraction of *Virtual Prototypes* (VPs), we unify HW/SW coverage by viewing the HW/SW system as a single model. This enables the measurement of structural HW/SW metrics like line, function, and branch coverage via a novel non-intrusive approach, where neither the VP (representing the HW) nor the SW requires any modification. Second, based on the unified HW/SW coverage, we introduce *relation coverage*. The innovation is that the user can define a relation between the frequency of executing lines in the SW and the execution count of corresponding lines of the HW model. This relation expresses expected behavior to be covered during testing. As a case study, we consider HW/SW testing of a Gyroscope sensor controlled by SW running on a RISC-V VP.**

I. Introduction

HW/SW systems are crucial because they form the backbone of modern computing and technological infrastructure. Designing HW/SW systems presents significant challenges, especially due to the non-trivial interaction between HW and SW needed to balance real-time requirements, power consumption, as well as to ensure robust performance in a range of applications. An industrial-proven approach to handle this challenge is the use of *Virtual Prototypes* (VPs) in the design process. In essence, a VP is a high-level, executable model of the entire HW platform, including processors, accelerators, sensors, actuators, buses, displays, etc., capable of running unmodified production SW [1]. VPs are created in SystemC, a standardized C++ class library (IEEE 1666, [2]), and they enable to parallelize SW and HW development [3]. VPs achieve significantly faster simulation performance compared to *Register Transfer Level* (RTL) through the use of *Transaction Level Modeling* (TLM) [4]. The key principle of TLM is to abstract the detailed implementation of signal-level interactions and timing specifics within the HW while the SW is run by an *Instruction Set Simulator* (ISS) that models the processor.

Clearly, HW/SW testing plays a critical role during system design because it ensures that the complex and interacting HW and SW components work correctly and reliably together. To measure the thoroughness of HW/SW tests, coverage metrics are employed [5]. However, the coverage metrics are typically only considered in isolation, i.e. there is no common established HW/SW metric (see related work below), which overall reduces the time advantage of VP-based HW/SW co-design.

Contribution: In this paper, we make two major contributions: The first is to view the VP code (representing the HW) and the SW (running on the ISS of the VP) as a single model which allows us to define unified HW/SW coverage metrics. More precisely, this includes the *structural HW/SW metrics*: *line coverage*, *function coverage*, and *branch coverage*. To measure these metrics during SystemC simulation, we devise a novel non-intrusive approach leveraging dynamic binary instrumentation such that neither the SW nor the VP have to be modified by the user. The second contribution builds on top of the proposed unified HW/SW coverage metrics and is denoted as *relation coverage*: The idea originates from the fact that we know the number of executions for a specific source code line (e.g. in SW) with our metrics, and can relate this number to a corresponding number at another source code location, e.g. HW. Very intuitive and valuable relations, motivated by complex HW/SW interactions, can be defined and checked.

Related Work: Coverage metrics are typically measured in isolation, some examples include [6]–[8] for SW and [9]–[12] for HW. Recently, in [13] an approach was presented which aims to integrate HW coverage and SW coverage to provide this information to a coverage-guided fuzzer closing the test generation loop. The approach is intrusive and tied to the test strategy, whereas our approach is non-intrusive and independent of the used test strategy. Finally, to the best of our knowledge, the proposed relation coverage is novel and provides a new way to describe valuable functional intend of complex HW/SW interactions.

Our experiments are based on the open-source RISC-V VP++ [14] where we integrate a SW-controlled Gyroscope sensor. We develop a test suite and show how the coverage evolves when adding more HW/SW tests. Moreover, we demonstrate that the innovative relation coverage is effective in revealing non-trivial bugs caused by HW/SW interactions.

II. Unified HW/SW Coverage

In this section, we define unified HW/SW coverage metrics. Viewing the SystemC VP, representing the HW, and the SW as a single model, we formalize the three structural metrics *line coverage*, *function coverage*, and *branch coverage*.

A. Running Example and Preliminary Considerations

We support the formalization by using a running example, a timer peripheral connected to a RISC-V VP. The timer peripheral is implemented in SystemC TLM-2.0 and contains the register `interval_reg` storing the time interval to trigger the next interrupt (e.g. every 5 milliseconds). The registers can be configured from SW which also handles the incoming interrupts.

Structural coverage is a metric that evaluates how often the code is executed by the test suite. As source code is stored in files, the most obvious way to split the source code is line by line. In general, a source code *line* can either be *executable* or *non-executable*. Examples for the latter case include comments, pre-processor statements, curly braces, declarations etc. In case of our timer example this holds for the complete code shown in Listing 1. Note that such code cannot be covered. For the metrics, which we formalize in the following, we are of course

```
1 // HW timer peripheral
2 #include <systemc>
3 #include <tlm_utils/simple_target_socket.h>
4 struct Timer : public sc_module {
5   uint interval_reg; // time interval register
6   uint irq_number; // interrupt index
7   socket<Timer> tsock;
8   interrupt_gateway *plic;
9   SC_HAS_PROCESS(Timer);
10 }
```

Listing 1: Not-executable declarative code excerpt of HW timer peripheral

```
12 // Constructur of HW timer peripheral
13 Timer(sc_module_name name, uint irq_number){
14   irq_number = irq_number;
15   tsock.register_b_transport(this, &transport);
16   SC_THREAD(run);
17 }
18
19 // Generating periodic interrupts
20 void run() {
21   while (true) {
22     wait(sc_time(interval_reg, SC_MS));
23     plic->gateway_trigger_interrupt(irq_number);
24   }
25 }
```

Listing 2: Executable code excerpt of HW timer per.; relevant for line coverage

interested in all executable source code lines, i.e. the ones which can be covered during simulation. We look again at the timer peripheral, but other code fragments. Again code of the HW side is shown in Listing 2: Line 13–17 will be executed when the timer is instantiated, sets the interrupt number, sets the TLM transport function called from the bus, and defines the SystemC thread `run()`. This `run()` thread models the behavior of the timer peripheral, i.e. triggering an interrupt after (Line 23) the specified amount of time (Line 22). The SW side is given in Listing 3. It depicts the main function of the SW. This example is simplified, i.e. after registering the interrupt handler function (Line 3) and configuring a time interval of 5 (Line 4), there is a loop which waits for 10 interrupts (Line 5, does nothing else), and then again the time interval is configured. To summarize, the last two exemplary listings illustrated executable source code lines from HW *and* SW. For such code, we define our HW/SW coverage metrics in the following.

B. Line Coverage

To formally capture the number of times a line has been executed or not during simulation, we assign an execution counter E to each line L. This means from now on when we talk about a line, we talk about a tuple consisting of the source code line and the counter. Formally, we have

$$(E, L)$$

Since non-executable code (comments, declarations etc. as described in the previous section) is excluded, we have

$$L_{All} = \{(E_i, L_i) | i \in I\}$$

where I contains only the indices of executable lines.

Finally, we can formalize HW/SW line coverage. Assume we have simulated a test suite, the execution counters of our HW/SW system given as L_{All} will be either zero or greater than zero. Using this information, we can create two subsets: L_U for uncovered (not executed) lines and L_C for covered (executed) lines. This can be expressed as:

$$L_U = \{(E_i, L_i) | i \in I \land E_i = 0\}$$
$$L_C = \{(E_i, L_i) | i \in I \land E_i > 0\}$$

```
1 // Simple SW configuring HW timer peripheral
2 int main() {
3   register_interrupt_handler(ID, timer_irqhandler);
4   set_interval_reg(5);
5   while(counter < 10) { asm volatile ("wfi"); }
6   set_interval_reg(UINT_MAX);
7   return 0;
8 }
```

Listing 3: Executable code excerpt of SW for timer per.; rel. for line coverage

```
27 // TLM transport called from TLM bus with
     transaction object
28 void transport(tlm::tlm_generic_payload &trans,
     sc_core::sc_time &delay) {
29   ...
30   if (trans.get_address() == 0x10){
31       interval_reg = *(trans.get_data_ptr());
32   } else {
33       ...
34   }
35   ...
36 }
```

Listing 4: Executable code excerpt of HW timer per.; rel. for branch coverage

Dividing now the cardinality of the set L_C by the cardinality of the union $L_U \cup L_C$, we get the HW/SW line coverage metric as:

$$LineCov = \frac{|L_C|}{|L_U \cup L_C|} \times 100$$

C. Function and Branch Coverage

For function coverage, we look again at our running example in Listing 2. In Line 16 the SystemC thread `run()` has been defined which is realized in SystemC as a function. Its implementation starts at Line 20. The function includes an infinite loop that activates the interrupt at regular intervals configured by the interval register. To check if a function is executed or not, it is sufficient to determine if the first line of the function was executed. This means by shrinking our index set I to just include the first lines of the functions instead of all lines, we can reuse the formalization we have introduced for line coverage.

For branch coverage, we use Listing 4 to illustrate the concept. The example contains the HW functionality to write the value into the interval register. Thereby the data is initially send from SW using memory-mapped I/O and hence finally routed into the HW timer peripheral via the TLM bus which passes it as transaction object to the transport function (Line 28). We now look at the branch in Line 30 of Listing 4 which checks if the interval register is addressed. When executing the branch, there are two possible paths. The first one is taken, if the branch condition is true. In this case Line 31 is executed. If the condition is false, Line 33 is executed. An appropriate index set I containing all lines for the true and false cases defines the branch coverage. Again, we can reuse the formalization we have introduced for line coverage to compute the final percentage.

III. Relation Coverage

As shown so far, for structural coverage it is common to use counters to determine whether code has been covered or not. This differentiation is binary, even though the counters can assume values beyond just 0 or 1. Consequently, the question emerges whether the extra information regarding the frequency of code execution can be utilized to enhance system robustness. The idea is to put the counters for different source code lines into relation. Since we have unified HW/SW metrics as introduced in the previous section, we can define these relations comprehensively.

```
10 void set_interval_reg(uint interval_ms) {
11    *INTERVAL_REG_ADDR = interval_ms;
12 }
13 // SW interrupt handler
14 void timer_irqhandler() {
15     counter++;
16 }
```

Listing 5: Executable code excerpt of SW for timer peripheral

A. Equal Relation

We start with a natural relation which requires the equality of the involved source code lines. Let us motivate this relation type using two examples, both reflect complex HW/SW interaction.

If we look at the SW code in Listing 5 and there at Line 11 as well as in Listing 4 of the HW side and there at Line 31, we can see that the purpose of these two lines is to write a value into a register. Assuming that no other line in SW is writing into the register, the number of executed memory-mapped I/O SW write (Line 11@Listing 5) has to be equal to the number of assignments to the register in HW (Line 31@Listing 4), in other words the respective executions counters match. If the counters are different, we have found a bug.

As second example for an equal relation we consider interrupts. The HW side triggers an interrupt (Line 23@Listing 2) which is then finally leads to the execution of the SW interrupt handler (Line 15@Listing 5) which increments a counter. If the number of triggered interrupts by the peripheral and the number of executed interrupt handlers for the peripheral don't match, there is a bug.

To summarize, an equal relation $\sim_=$ captures that the equality of the executions counters of both user-defined lines L_i and L_j is required:

$$(E_i, L_i) \sim_= (E_j, L_j) :\Leftrightarrow E_i = E_j$$

Before we discuss the computation of the coverage metric, we expand this idea of relation coverage further by utilizing alternative relations.

B. Greater or Equal Relation

An alternative to the equal relation is the greater or equal relation. The motivation for such relations can be shown on the example of Listing 3 Line 5, where the SW waits for the interrupt and Listing 2 Line 23 where the HW is triggering the interrupt. In this case it is required that the interrupt is triggered within the HW at least as often as the SW is waiting for it. If this is not the case, the SW does not further execute (is stuck).

These type of relations can be formulated as:

$$(E_i, L_i) \sim_\geq (E_j, L_j) :\Leftrightarrow E_i \geq E_j$$

C. Sum Relation

The third relation is the sum relation as it sums up execution counters from more than two source code lines. For the equal relation we have assumed that Line 11 from Listing 5 is the only position where the interval register is written. Looking at Listing 3 on Line 4 and Line 6 we can see that the main function is executing two times the function which writes into the register. We can now state that the counter in the *set_interval_reg* function is required to be sum of the counters on the two lines within the *main* function. Additionally, the sum of these counters has to match the read interval register execution counter within the HW from Line 31 in Listing 4. In general, such a relation can be expressed as:

$$(E_i, L_i) \sim_\Sigma \sum_{j \in J} (E_j, L_j) :\Leftrightarrow E_i = \sum_{j \in J} E_j$$

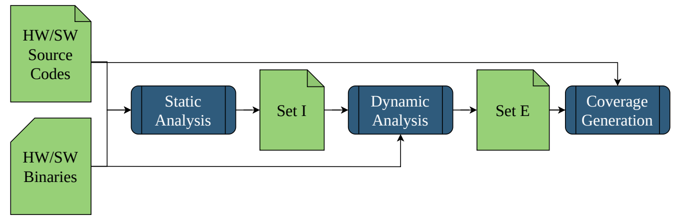

Fig. 1: Non-intrusive coverage measurement flow

where J contains the line indices where the respective counters are added up.

D. Relation Coverage Metrics

Similar to the introduced structural coverage formalization, it is now possible to create two sets: $\sim_C$ for covered relations and $\sim_U$ for uncovered relations. This makes it possible to calculate the ratio between the cardinalities of the covered and uncovered relation sets:

$$RelationCov = \frac{| \sim_C |}{| \sim_U \cup \sim_C |} \times 100$$

where $\sim$ is replaced by $\sim_=$, $\sim_\geq$, or $\sim_\Sigma$, respectively.

IV. Non-Intrusive Coverage Measurement

In this section, we focus on the implementation of the proposed HW/SW coverage metrics. To non-intrusively measure the coverage during SystemC simulation, we extend the approach from [15]. The main idea of [15] is to utilize the *Host-to-SW memory hierarchy* for dynamic runtime instrumentation based on *Program Counter* (PC) addresses from (a) the host (denoted as PC_{host}) which runs the VP simulation binary, and (b) the ISS which contains the PC_{HW} representing the PC of the ISS pointing to the running SW of the simulated HW/SW system. These PC addresses of interest are directly translated when compiling the VP source code and cross-compiling the SW source code using the debug information. As shown in Fig. 1, we kept the overall flow of [15] but modified the inner workings. The first part changed was the static analysis to identify all lines L_i where we need to add the execution counters E_i. These lines are translated into the *Set I* file containing the addresses of all executable lines L_i. This address set is then used in the dynamic analysis. The dynamic analysis is based on dynamic runtime instrumentation which allows that both, the HW (the VP) and the SW to remain unmodified. This step generates the *Set E* file containing all execution counters E_i for the corresponding addresses. This set is then processed by the coverage generation to output the coverage metrics as defined in Section II and Section III, respectively.

V. MEMS Gyroscope Experiment

We applied our approach to assist the development of a test suite for a *Micro Electromechanical System* (MEMS) Gyroscope HW peripheral and SW library to use the peripheral. The HW of our experiment is based on the open-source *RISC-V VP++* [14][1]. This VP was extended with a MEMS Gyroscope peripheral generating data for the x-, y-, and z-axes. The peripheral is additionally connected to the PLIC interrupt controller of the VP. The SW contains the library which should be tested. This library consists of functions designed to interact with the Gyroscope.

To develop the test suite for our experiment we took a coverage-guided approach, meaning we executed an empty test suite and

[1] https://github.com/ics.jku/riscv-vp-plusplus

TABLE I: Coverage for test suites in %, where light blue represents structural coverage and green represents relation coverage

	A.0	B.0	C.0	C.1	C.2
Function	16	100	100	100	100
Line	18	88	100	100	100
Branch	12	70	100	100	100
$\sim_=$	0	40	40	80	100
$\sim_\geq$	0	0	100	100	100
$\sim_\Sigma$	0	0	100	100	100

TABLE II: Coverage of relations are named according to the schema <LHS>@<HW/SW><$\sim_{type}$><RHS>@<HW/SW> where "x" means not covered and "✓" means covered.

	A.0	B.0	C.0	C.1	C.2
RegXR@SW$\sim_=$RegXR@HW	x	✓	✓	✓	✓
RegYR@SW$\sim_=$RegYR@HW	x	x	x	✓	✓
RegZR@SW$\sim_=$RegZR@HW	x	x	x	✓	✓
Constr@HW$\sim_=$Init@SW	x	✓	✓	✓	✓
IRQT@HW$\sim_=$IRQH@SW	x	x	x	x	✓
SampleX@HW$\sim_\geq$RegX@SW	x	x	✓	✓	✓
SampleY@HW$\sim_\geq$RegY@SW	x	x	✓	✓	✓
SampleZ@HW$\sim_\geq$RegZ@SW	x	x	✓	✓	✓
RegCR@HW$\sim_\Sigma$RegCR@SW	x	x	✓	✓	✓
RegCW@HW$\sim_\Sigma$RegCW@SW	x	x	✓	✓	✓
TLMRW@HW$\sim_\Sigma$RegRW@SW	x	x	✓	✓	✓
TLMR@HW$\sim_\Sigma$RegR@SW	x	x	✓	✓	✓
TLMW@HW$\sim_\Sigma$RegW@SW	x	x	✓	✓	✓

then added tests until full coverage was reached. For comprehensibility, each development step of the test suite has a version number containing a major (alphabet) and a minor (number) version. The major version was increased when the test suite was extended. The minor version was increased when a bug fix was applied, e.g. $B.2$ means major version B and minor version 2.

The coverage results for each iteration are shown in Table I. The columns contain the aforementioned versions of the test suite. The lines within the table are the collected coverages, where *Function*, *Line*, *Branch* are the proposed structural HW/SW coverage metrics and $=$, $\geq$ and Σ are the relation metrics. Table II shows the results for the single relations which were derived for the MEMS Gyroscope HW/SW. In the following, we describe each major and minor step in development.

a) $A.0$ to $B.0$: As already said, the initial test suite $A.0$ was empty. This means that nothing from the MEMS Gyroscope SW library is executed. On the HW side, the sensor is already included in this step which means the constructor and the sample thread are already executed. This explains the low but existing structural coverage of $12-18\%$ for the initial package. The next step was to reach full function coverage. This was done by enabling the interrupt and calling each SW function from the test suite.

b) $B.0$ to $C.0$: Table I shows that for test suite $B.0$ the line coverage is 88% and the branch coverage is 70%. The next step was to introduce additional tests to reach 100% for these two coverage metrics. These tests included different combinations of the function calls and different amounts of how often the data was read from the peripheral. This resulted in 100% for line and branch coverage.

c) $C.0$ to $C.1$: For test suite $C.0$ we can see that for the equal relation only 40% have been achieved, meaning, although we have reached full structural coverage, we have uncovered relations. Looking at Table II, we can see that the relations, concerning the $y-$ and $z-$axis register reads, are not covered. As we have designed the tests to ensure that each axis undergoes

```
98  if(int_enabled) {
87    plic->gateway_trigger_interrupt(irq_number);
    }
```

Listing 6: Interrupt HW trigger for $C.2$; numbers give execution counts

```
68  void mems_gyro_irq_hanlder() {
68    has_data = 1;
    }
```

Listing 7: Interrupt SW handler for $C.2$; numbers give execution counts

a different number of read operations, we have found that the address of the register y and the register z have been swapped. Fixing this lead to 80% coverage for the equal relations.

d) $C.1$ to $C.2$: The last not covered relation for our test suite checks if the interrupt triggered in HW and the calls to the interrupt handler in SW match. If this is not the case, we are loosing interrupts. When looking into the HW coverage (Listing 6) we see the interrupt being triggered 87 times. The interrupt handler from the SW Listing 7 shows only 68 executions. A possible reason is that the sampling rate is too high. Setting the proper sampling rate lead to full coverage, completing the development.

VI. Conclusions

In this paper, we first unified HW/SW coverage and defined structural HW/SW coverage metrics leveraging the abstraction of virtual prototypes. On top of the structural metrics, we devised the novel relation coverage metric. Essentially, relations between different source code locations can be captured which allow in particular to reflect complex HW/SW interactions. We have additionally developed a framework to measure the proposed coverage in a non-intrusive way, i.e. neither the VP nor the SW has to be modified. We demonstrated the value of our metrics for a MEMS Gyroscope sensor. In particular, the innovative relation coverage helped to easily find non-trivial HW/SW interaction bugs.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

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

[2] *IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Std. 1666 (Revision of IEEE Std 1666-2011), 2023.

[3] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

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

[5] A. Piziali, *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[6] D. Lettnin *et al.*, "Coverage driven verification applied to embedded software," in *ISVLSI*, 2007, pp. 159–164.

[7] J. C. Costa and J. C. Monteiro, "Coverage-directed observability-based validation for embedded software," *TODAES*, vol. 18, no. 2, Apr 2013.

[8] Kruse *et al.*, "A highly configurable test system for evolutionary black-box testing of embedded systems," in *GECCO*, 2009, p. 1545–1552.

[9] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *D&T*, vol. 18, no. 4, pp. 36–45, 2001.

[10] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *TVLSI*, vol. 14, no. 1, pp. 57–68, 2006.

[11] D. Große, H. Peraza, W. Klingauf, and R. Drechsler, "Measuring the quality of a SystemC testbench by using code coverage techniques," in *FDL*, 2007, pp. 146–151.

[12] D. Große, U. Kühne, and R. Drechsler, "Estimating functional coverage in bounded model checking," in *DATE*, 2007, pp. 1176–1181.

[13] N. Bruns, V. Herdt, and R. Drechsler, "Unified hw/sw coverage: A novel metric to boost coverage-guided fuzzing for virtual prototype based hw/sw co-verification," in *FDL*, 2022, pp. 1–8.

[14] M. Schlägl, C. Hazott, and D. Große, "RISC-V VP++: Next generation open-source virtual prototype," in *Workshop on Open-Source Design Automation*, 2024.

[15] C. Hazott and D. Große, "DSA monitoring framework for HW/SW partitioning of application kernels leveraging VPs," in *DVCon Europe*, 2023, pp. 34–41.