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*

Vladimir Herdt[1] Hoang M. Le[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{vherdt,hle,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Electronic systems integrate an increasingly large number of components on a single chip. This leads to increased risk of faults, e.g. due to radiation, aging etc. Such a fault can lead to an observable error and failure of the system. Therefore, an error effect simulation is important to ensure the robustness and safety of these systems. Error effect simulation with Virtual Prototypes (VPs) is much faster than with RTL designs due to less modeling details at TLM. However, for the same reason, the simulation results with VP might be significantly less accurate compared to RTL. To improve the quality of a TLM error effect simulation, a fault correspondence analysis between both abstraction levels is required. This paper presents a case study on applying fault localization methods based on symbolic simulation to identify corresponding TLM errors for transient bit flips at RTL. First results for the interrupt controller of the SoCRocket VP, which is being used by the European Space Agency, demonstrate the applicability of our approach.**

I. Introduction

Ensuring the functional safety of electronic systems becomes one of the most important issues nowadays, as these systems are being more and more deeply integrated into our lives. Even if a system can be proved to perform its intended functionality correctly, failures are still possible due to hardware (HW) faults caused e.g. by radiation or aging. The risk of such faults is rapidly increasing with the raising complexity of design and technology scaling. To evaluate and facilitate the development of safety measures, fault injection is a widely accepted approach, which is also recommended in different functional safety norms such as IEC 61508 and ISO 26262.

Traditional HW fault injection approaches operate on low levels of abstraction such as gate level or *Register Transfer Level* (RTL). While physical faults can be quite accurately modeled at these abstraction levels, the slow simulation speed becomes a major bottleneck for modern systems. This used to be a problem for software (SW) development and system verification as well, until the emergence of *SystemC Virtual Prototypes* (VPs). VPs are basically full functional SW models of HW abstracting away micro-architectural details. This higher level of abstraction, often termed as *Transaction-Level Modeling* (TLM) [1], allows significantly faster simulation compared to RTL. Therefore, safety evaluation using VP-based fault injection, envisioned as error effect simulation in [2], is a very promising direction.

Fault injection techniques for SystemC have attracted a large number of work, see e.g. [3], [4], [5], [6]. They form a technical basis for error effect simulation, but do not focus on the core problem: the higher level of abstraction of TLM poses a big challenge in error modeling. Please note that we use the term "fault" for RTL and "error" for TLM due to the fact that TLM is just a modeling abstraction. A high-level error model, if not carefully designed, would yield significantly different simulation results compared to a low-level fault model [7], [8]. This would make the safety evaluation results using VPs to become misleading. Unfortunately, deriving an accurate high-level error model is very difficult [7], [8].

In this paper, we examine the idea of a novel cross-level fault correspondence analysis to aid the design of such error model. The prerequisite of our analysis is the availability of an RTL model and its corresponding TLM model. Then, for a given RTL fault model, our analysis automatically identifies for an RTL fault a set of candidates for error injection in the TLM model. These candidates are potentially *equivalent* to the RTL fault, in the sense that the error-injected TLM model would produce the same failure as the fault-injected RTL model.

The core idea of this RTL-to-TLM fault correspondence analysis is inspired by the concept of *formal fault localization* (see [9] for C and [10] for SystemC TLM). These approaches automatically identify candidates for modifications in the model under verification, which would make a given set of failing testcases to become passing testcases. Such a candidate potentially points to the location of the bug that causes a failure. Our analysis is dual to this: Given successful TLM simulations[1], we search for locations to inject an error to

* This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

[1]A successful TLM simulation produces the same output as RTL simulation under the same inputs.

produce the same failure observed in faulty RTL simulations. Hence, we can apply the same set of techniques: instrumenting the TLM model to include non-deterministic errors and leveraging existing TLM model checkers to compute the candidates.

The paper also presents first results of a case study on the *Interrupt Controller for Multiple Processors* (IRQMP) of the SoCRocket VP, which is being used by the European Space Agency [11], to demonstrate the feasibility of the analysis. In particular, we implement the cross-level analysis on top of the recent symbolic simulation approach for SystemC [12], [13] and apply our analysis to find TLM error injection candidates for transient bit flips at RTL. To the best of our knowledge, it is the first time such results on fault correspondence between TLM and RTL are reported.

II. Related Work

As mentioned earlier, a large number of fault injection techniques for SystemC TLM models exists. These approaches assume some TLM error models without qualitative or quantitative assessment of their correspondence to RTL faults. [14] includes a comprehensive list of TLM errors that might result from RTL bit flips. The paper also provides a set of guidelines on how to (manually) derive corresponding TLM errors, whereas our analysis is automated.

Another line of work is mutation analysis for SystemC TLM models [15], [16], [17]. At the heart of any mutation analysis is also an error model. However, the purpose of such model is to mimic common design errors, but not HW faults caused by external impact.

The most close work to ours is the one in [18], which proposes an automatic RTL-to-TLM transformation to speed up RTL fault simulation. The transformation produces an equivalent TLM model from each fault-injected RTL model. The obtained results can be mapped back to RTL. However, this approach relies on a particular RTL-to-TLM transformation. Such transformation might not provide the best possible speed-up compared to hand-crafted TLM models. In contrast to our analysis, this approach is not applicable when the corresponding TLM model already exists.

III. Preliminaries

In this section we first briefly describe the interrupt controller IRQMP of the SoCRocket VP used later in the case study. The second half of this section briefly describes the (extended) intermediate verification language to enable the formal representation of TLM models as well as leveraging advanced symbolic execution techniques.

A. Interrupt Controller for Multiple Processors (IRQMP)

The IRQMP is an interrupt controller from the SoCRocket VP supporting up to 16 processors [11]. It consists of a register file, several input and output wires and an APB Slave bus interface for register access. The register file contains shared and processor-specific registers. Every register has a bit width of 32. Each bit naturally represents an interrupt line.

The IRQMP supports incoming interrupts (using the *irq_in* wire or *force* register) numbered from 1 to 31 (interrupt line 0 is reserved/unused). Lines 15:1 are regular interrupts and lines 31:16 are extended interrupts. In regular operation mode, IRQMP ignores all incoming extended interrupts. The *irq_req* and *irq_ack* wires are connected with every processor and allow to send interrupt requests and receive acknowledgements.

The functionality of the IRQMP is to process incoming interrupts by applying masking and prioritization of interrupts for every processor. Prioritization of multiple available interrupts is resolved using the *level* register. A high (low) bit in the *level* register defines a high (low) priority for the corresponding interrupt line. On the same priority level, interrupt with larger line number is higher prioritized. See the specification [19] of the IRQMP for more details.

B. Extended Intermediate Verification Language (XIVL)

The XIVL has been proposed in [13] as an extension to the SystemC *Intermediate Verification Language* (IVL) [12] to act as high-level intermediate representation with formal verification support for SystemC TLM. In essence, the XIVL provides a small core of instructions to capture the cooperative multi-threaded simulation semantics of SystemC and supports all arithmetic and logic operators of C++. For verification purposes it provides symbolic expressions as well as the assume and assert functions with their usual semantics. Control flow is modeled using high level control flow structures. A small set of object oriented programming features - including virtual functions, inheritance and dynamic dispatch - is supported for modeling TLM designs more naturally.

Symbolic simulation for SystemC as proposed in [12], [20], [21], [22] essentially combines symbolic execution with complete exploration of all process schedules. Partial Order Reduction techniques are employed to improve scalability by pruning redundant schedules [23], [24].

A formal verification approach based on symbolic simulation has been presented in [13] and demonstrated for the IRQMP TLM model. In this paper, we instrument the available TLM model to include non-deterministic errors and leverage the existing TLM model checker based on symbolic simulation for fault localization.

IV. RTL-to-TLM Fault Correspondence Analysis

Our proposed RTL-to-TLM fault correspondence analysis allows to find errors in a TLM model that correspond to faults in the RTL model. We assume that the RTL and TLM model are functionally equivalent, i.e. they produce the same outputs when given the same inputs. Furthermore, we assume that a set of (representative) inputs, e.g. in the form of testcases, is available for the RTL or TLM (since both use the same inputs) model. These inputs should preferably cover a large set of functionality of the design. Similarly, we assume that a set of fault injection locations is available for the RTL model. Otherwise, the fault injection locations can be obtained by tracing the execution of the RTL model, e.g. using an observer class, based on the available testcases. Please note that a fault injection location consists of three pieces of information: 1) a source line, 2) an injection time, i.e. a number that denotes which execution of this source line should be fault injected

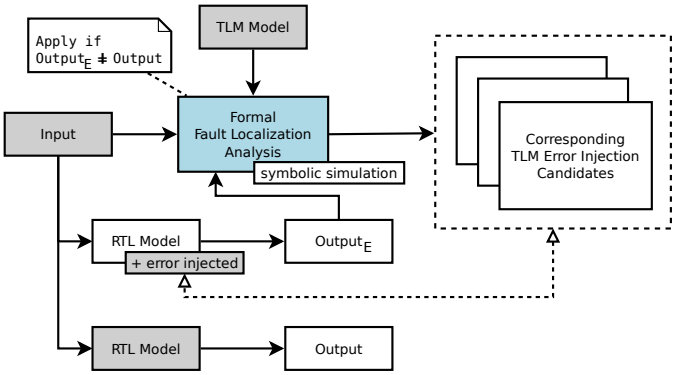

Fig. 1. Fault correspondence analysis overview

and 3) the bit position which shall be flipped. The reason for information (2) is that we consider transient faults in this paper.

A. Correspondence Analysis Overview and Algorithm

Fig. 1 shows an overview of our analysis approach. A corresponding algorithm is shown in Fig. 2. It computes a set of corresponding error candidates on TLM level for every injected fault on RTL. The algorithm considers every fault injection location L on the RTL model. First we construct the faulty RTL model with respect to L (Line 4). In our implementation, we attach an fault injection class as observer to our RTL model that can inject an error at runtime. Then we simulate the correct and faulty RTL model with the same input. There are two possible cases: (i) Both RTL models produce the same output. In this case the injected fault had no observable effect and simulation is repeated with a different input (Line 5). (ii) Otherwise, both RTL models produce a different output. Then we apply a formal fault localization analysis based on symbolic simulation on the TLM model (Line 9) to produce a set of possible corresponding error locations C on the TLM model. Essentially, all these reported error locations C produce the same failure, i.e. same output, as the faulty RTL model for the given input. The set C is integrated into the result set for L by computing a set intersection (Line 13) - or simple assignment in case C is the first result (Line 11). If the result set is empty, our analysis concludes that no corresponding TLM error can be found for the current RTL fault and we consider the next RTL fault. The reason is that a corresponding TLM error must produce the same output as the faulty RTL model for *all* inputs. Otherwise, the result set is not empty, we continue with the next input.

B. Example

a) Method Overview: As an example, consider a bit flip fault in the RTL model of the IRQMP (interrupt controller, see preliminaries Section III-A) when initially configuring the mask register using a bus transfer operation as fault injection location L. Furthermore, consider three test scenarios with different inputs:

1) Send interrupt using the *irq_in* (incoming interrupt) wire
2) Send interrupt using the *force* register

```
1  result ← ∅ /* Mapping from RTL fault injection
     location to set of corresponding TLM error
     injection candidates                          */
2  for L ∈ RTL-fault-injection-locations do
       /* Start with empty result set for L       */
3      result[L] ← ∅
4      RTL-Model_E ← RTL-Model with L injected
5      for input ∈ testcases do
           /* Check if RTL fault L has observable
              effect                               */
6          output ← simulate(RTL-Model, input)
7          output_E ← simulate(RTL-Model_E, input)
8          if output ≠ output_E then
9              candidates ← fault-localization-analysis(
                            TLM-Model, input, output_E)
10             if result[L] = ∅ then
                   /* First set of candidates      */
11                 result[L] ← candidates
12             else
                   /* Combine with existing set    */
13                 result[L] ← result[L] ∩ candidates
14             if result[L] = ∅ then
                   /* No corresponding TLM error for L
                      found, check next RTL fault  */
15                 break
```

Fig. 2. Fault correspondance analysis

TABLE I
CORRESPONDING TLM ERROR INJECTION CANDIDATES
(CORRESPONDING ERROR HIGHLIGHTED)

TLM Error Injection Location	Inputs		
	1	2	3
Bus Transfer			
Mask Register Configuration	X	X	X
Force Register Configuration		X	
Wire Transfer			
Incoming Interrupt Wire	X		
Computation			
Prioritization Logic 1	X	X	
Prioritization Logic 2	X	X	

3) Send the same interrupt twice using the *irq_in* wire

All of these inputs result in different outputs for the RTL model with and without injection of the fault L. In particular no interrupt is generated by the interrupt controller for the masked interrupt line. On the TLM model, our analysis identifies different candidates for corresponding errors. In particular, it identifies different transient bit flip errors during computations as well as wire/bus transfer that lead to the same observable behavior.

The results are summarized in Table I. For the first and second input we obtain multiple possible error locations in the TLM model. By sending the same interrupt twice (third input) to the interrupt controller, the effects of a transient computation error and non-related transfer error are eliminated. By computing the intersection of all possible error locations for these inputs, the corresponding TLM error is obtained - bus transfer error during configuration of the mask register.

```
1  -- combinatorial VHDL process, which essentially contains the whole logic of the IRQMP - sensitive on the reset
       signal, update of internal signals as well as incoming bus and interrupt signals
2  comb : process(...)
3    -- send out current internal signal values and compute new values which will overwrite the current values in the
         next clock cycle
4    variable v  : reg_type;  -- registers to store regular interrupt lines for local computation
5    variable v2 : ereg_type; -- registers to store extended interrupt lines for local computation
6    begin
7      -- ... prioritize interrupts, register read ...
8
9      -- register write
10     if ((apbi.psel(pindex) and apbi.penable and apbi.pwrite) = '1' and
11         (irqmap = 0 or apbi.paddr(9) = '0')) then -- essentially, check that bus is enabled and used in write mode
12       case apbi.paddr(7 downto 6) is -- decode target register
13       -- ...
14       when "01" => -- write to processor specific mask register
15         for i in 0 to ncpu-1 loop -- iterate over all processors
16           if i = conv_integer( apbi.paddr(5 downto 2)) then -- decode and check target processor
17             v.imask(i) := apbi.pwdata(15 downto 1); -- write to mask of processor i, RTL fault injected here
18             if eirq /= 0 then -- check if extended interrupts are also handled
19               v2.imask(i) := apbi.pwdata(31 downto 16); -- in this case also update the extended interrupt lines of
                     the mask register
20             end if;
21           end if;
22         end loop;
23       -- ...
24       end case;
25     end if;
26
27     -- ... register new interrupts, interrupt acknowledge, reset ...
28 end process;
```

Fig. 3. Example IRQMP code excerpt in VHDL showing register configuration, to illustrate fault injection on RTL (line with fault injection highlighted)

```
1  // generic (using templates) and re-usable TLM
       register class, the inherited class provides the
       basic interface and some default implementation
2  template<typename DATA_TYPE>
3  class sr_register : public sc_register_b<DATA_TYPE> {
4    //...
5  public:
6    void bus_write(DATA_TYPE i) {
7      // callbacks notify observers about register
           access directly without context switches
8      raise_callback(SR_PRE_WRITE);
9      // corresponding TLM error location - update the
           internal register value, the write mask
           allows selective updates
10     this->write(i & m_write_mask);
11     // the IRQMP will trigger interrupt re-computation
           after the mask register is updated
12     raise_callback(SR_POST_WRITE);
13   }
14   //...
15 }
16
17 // a register bank groups multiple registers -
       similarly, this is a generic implementation
18 template<typename ADDR_TYPE, typename DATA_TYPE>
19 class sr_register_bank : public
       sc_register_bank<ADDR_TYPE, DATA_TYPE> {
20   typedef typename std::map<ADDR_TYPE,
         sr_register<DATA_TYPE> *> register_map_t;
21   register_map_t m_register; // use a mapping (address
         to register) to store registers
22   //...
23 public:
24   // bus read/write transactions matching a register
         address are automatically redirected to this
         class
25   bool bus_write(ADDR_TYPE offset, DATA_TYPE val) {
26     sr_register<DATA_TYPE> *reg =
           get_sr_register(offset); // retrieve register
           for the given (bus) address
27     if(reg) {
28       reg->bus_write(val); // update register value
29     }
30     return true;
31   }
32   //...
33 }
34
35 // register bank used as member variable in the IRQMP
36 sr_register_bank<unsigned int, unsigned int> r;
```

Fig. 4. TLM code for register configuration, showing a corresponding TLM error (line highlighted) for Fig. 3

b) Fault Correspondence: To further illustrate this fault injection example, Fig. 3 and Fig. 4 show relevant code of the IRQMP for configuring the *mask* register at RTL and TLM, respectively. The source line where the fault has been injected at RTL and the corresponding error location at TLM are highlighted. The RTL code is available in VHDL, and the TLM code in SystemC.

The RTL code stores internal signals for registers separated for regular (lines 15:1, *reg* type) and extended interrupts (lines 31:16, *ereg* type). The processing logic of the IRQMP is available in the combinatorial process *comb*, shown in Fig. 3. Essentially, it contains the whole logic of the RTL model and is triggered whenever some input or internal signal changes. It is responsible for interrupt prioritization, processing of register read/write requests and interrupt acknowledgements, and writing output signals. Internal signal values are updated in a separate process at every clock cycle. In particular Fig. 3 shows processing code for a bus write request to the CPU specific *mask* register for normal (Line 17) and extended interrupts (Line 19). The target register and processor are encoded in the bus address signal, they are decoded in Line 12 and Line 16, respectively. In this example a fault is injected in Line 17 during *mask* register configuration.

The TLM implementation, shown in Fig. 4, of the IRQMP keeps a register bank (Line 36), which essentially contains a mapping of an address value to a register (Line 21). A

bus write transaction will call the *bus_write* function of the register bank (Line 25-31). The function will retrieve the target register directly based on the write address in Line 26 and dispatch the write access to the register class in Line 28. The register will finally update its internal value in Line 10, which is the corresponding TLM error. Before and after the update, callback functions are used to notify the IRQMP and update its internal state directly without any context switches. In this case the main processing thread of the IRQMP will be notified to re-compute outgoing interrupts. Please note that the register bank implementation is not specific to the IRQMP but a generic and re-usable implementation. Furthermore, the TLM model can update the whole register, while the RTL model only updates bits 15:1 when extended interrupts are ignored. This is not a problem, since the prioritization logic of the TLM model simply ignores the extended interrupt lines in this case and therefore produces the same failure (when the corresponding error is injected) as the faulty RTL model.

V. Formal Fault Localization Analysis

This section describes our formal fault localization analysis, to obtain candidates for corresponding TLM errors, in more detail. We reduce this problem to a verification problem of assertion violations by encoding error injection selection non-deterministically and adding appropriate constrains to prune invalid solutions. Then we employ symbolic simulation for an efficient exhaustive exploration to find all possible solutions. In general different formal verification techniques besides symbolic simulation could also be used to find solutions.

Fig. 5 shows an overview of our approach. The analysis requires the TLM model as well as the input and output of the faulty RTL model (marked grey in Fig. 5). We assume that the TLM model is available in the XIVL format to apply formal analysis techniques. The TLM model contains annotations for a fine grained selection of instruction where error injection can take place (see 1 in Fig. 5).

First a testbench is generated by using the input and output to construct an input driver and result monitor, respectively (see 3 in Fig. 5). The testbench is also available in the XIVL format and contains the simulation entrypoint - the main function - which is responsible to setup all components. The result monitor contains the assertions which constrain valid solution.

Then the TLM model and testbench are automatically combined to a complete TLM model. During this process, the TLM model will be instrumented with symbolic error injection logic to non-deterministically select an error injection location for a transient one bit error (see 3 in Fig. 5). The annotations on the TLM model are used to guide the instrumentation.

Finally, the complete TLM model is passed to our symbolic simulation engine for a formal analysis. Based on the symbolic error injection logic instrumented into the TLM model, the symbolic simulation will find and report all concrete error injection locations that will cause the TLM model to produce the same failure, i.e. same output, as the faulty RTL model for the given input.

In the following we will discuss 1) annotations, 2) symbolic error injection logic, and 3) testbench encoding in more detail.

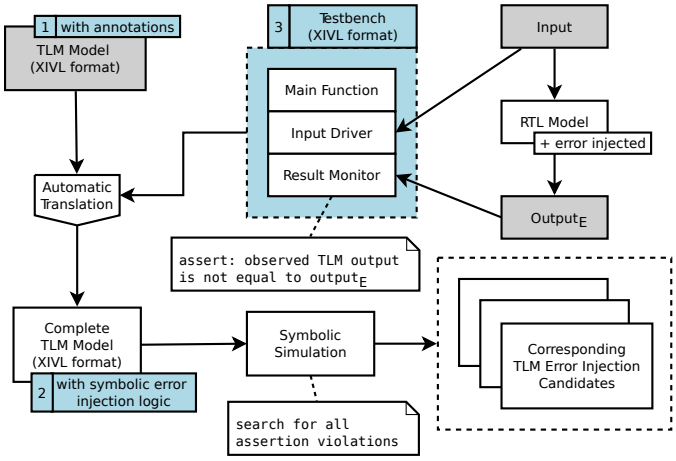

Fig. 5. Formal fault localization analysis overview

A. Annotations

We use annotations in the TLM model for a fine grained control of error injection. Assignment instructions are annotated to denote, that error injection can take place. During instrumentation, every annotated assignment will be modified to either stay unchanged or toggle a bit flip in the result - based on a non-deterministic choice during analysis (at runtime). For convenience, we also support function annotations. These will be propagated to all assignments in the function. Using annotations is a flexible approach to control error injection more precisely. These annotations can happen manually, or using a static analysis that modifies the code automatically, e.g. based on a specification provided by the user. This ensure that meaningful locations for error injection are reported - otherwise the initial state, or the output values would be modified. Furthermore, injection can be selectively activated and deactivated at runtime, based on a boolean global variable. This allows to run the same code blocks, e.g. functions, with and without error injection. We use it to deactivate error injection during initialization of the TLM model. The reason is that the same code is also used during testbench specific configuration, where error injection should be allowed.

We use a re-usable modeling layer for registers and wires based on [13], which are used by many TLM peripheral models including the IRQMP. Error injection in bus and wire transfer operations, as well as many computations can be handled by injecting errors in the modeling layer itself. An example for the register class is shown in Fig. 6. The *injectable* annotation on the *set_bit* function is propagated to both assignments.

B. Symbolic Error Injection Logic

We integrate a set of global variables and functions, shown in Fig. 7, into the complete TLM model for symbolic error injection. In particular the functions *inject_bitvector_error* and *inject_bool_error* are used from within the TLM model. Consider again the example in Fig. 6. For example, the assignment *@injectable this->value = value;* in the *write*

```
1  // XIVL implementation of the TLM register class
2  struct Register {
3    // write mask allows selective updates on bus access
4    uint32_t value;
5    uint32_t write_mask;
6
7    bool get_bit(Register *this, uint32_t index) {
8      return this->value & (1 << index);
9    }
10
11   @injectable
12   void set_bit(Register *this, uint32_t index, bool
         value) {
13     // injectable annotation is automatically
           propagated to both assignments
14     if (value) {
15       // set bit
16       this->value |= 1 << index;
17     } else {
18       // clear bit
19       this->value &= ~(1 << index);
20     }
21   }
22
23   uint32_t read(Register *this) {
24     return this->value;
25   }
26
27   void write(Register *this, uint32_t value) {
28     // errors can be injected at assignments marked
           with @injectable
29     @injectable this->value = value;
30   }
31
32   void bus_write(Register *this, uint32_t value) {
33     // ... PRE/POST write callback handling omitted in
           this example ...
34     @injectable this->value = value & write_mask;
35   }
36 }
```

Fig. 6. Excerpt of an annotated TLM register class in XIVL

function in Line 29 is transformed to *this->value = inject_bitvector_error(value, 32);* when annotations are resolved during instrumentation.

The *inject_bitvector_error* function (defined in Fig. 7) expects two arguments, an integer and its bitwidth. The bitwidth argument denotes the range of bits from which one is selected non-deterministically for flipping. The bitwidth argument is automatically generated based on static type informations during the instrumentation process, which rewrites the annotations. By using a 64 bit integer type as argument and return value, we also automatically support all integer types with smaller bitwidth. For this case study, 32 bit values are sufficient. The *inject_bool_error* function in principle works analogously.

For convenience we use short names in Fig. 7 for the global variables, in the TLM model they have a unique name prefix to avoid name clashes with existing code. The boolean *active* variable allows to selectively toggle error injection and thus control it more precisely. It is accessed by means of the *activate/deactivate_injection* functions defined in Line 8 and Line 4, respectively.

The *condition* variable is initialized with a symbolic integer value in Line 15 and together with the *id* variable allows non-deterministic selection of an injection location. This works as follows: Consider an invocation of *inject_bitvector_error* and assume that *active* is true. If no error has been injected

```
1  // dynamically toggle error injection
2  bool active = false;
3
4  void deactivate_injection() {
5    active = false;
6  }
7
8  void activate_injection() {
9    active = true;
10 }
11
12 // variables store injection choices for inspection
       and ensure that only single error is injected
13 int id = 0;
14 int location = -1;
15 int condition = ?(int);
16 int bit = -1;
17
18 int64_t flip_single_bitvector_bit(int64_t val, uint8_t
       bitwidth) {
19   // non-deterministically choose a single bit in
         *bitwidth* to flip
20   uint8_t x = ?(uint8_t);
21   assume (x >= 0 && x < bitwidth);
22   // perform the actual bit flip
23   val = val ^ (1 << x);
24   // store choice for later inspection
25   bit = x;
26   // result is symbolic due to non-deterministic choice
27   return val;
28 }
29
30 int64_t inject_bitvector_error(int64_t val, uint8_t
       bitwidth) {
31   // unique id ensures only a single error is injected
32   id += 1;
33   if ((condition == id) && active) {
34     // record the injection choice for later inspection
35     @track "one bit error injection";
36     location = id;
37
38     // perform a non-deterministic bit flip
39     val = flip_single_bitvector_bit(val, bitwidth);
40   }
41   return val;
42 }
43
44 bool inject_bool_error(bool val) {
45   // essentially similar to injecting a bitvector error
46   id += 1;
47   if ((condition == id) && active) {
48     @track "boolean error injection";
49     location = id;
50
51     val = !val;
52   }
53   return val;
54 }
```

Fig. 7. Encoding details for transient one bit error injection

yet, then both branch directions in Line 33 are feasible, i.e. *condition* $=$ *id* can evaluate to *true* and *false*. Therefore, symbolic execution will split into two independent paths S_T and S_F, respectively and explore both branch directions. This will update the path conditions of S_T and S_F with *condition* $=$ *id* and *condition* $\neq$ *id*, respectively. Please note, that *id* is a concrete integer value in this case, e.g. the number 4. Since *id* is incremented on every call of *inject_bitvector_error*, the true branch of the *if* statement in Line 33 becomes infeasible for the S_T path and all its descendants. Therefore, at most a single error is injected on every execution path. The *location* variable stores a copy of the injection *id* for debugging purposes.

The *@track* instruction is specifically recognized by our symbolic simulation engine and records a snapshot of all

instruction pointers of the callstack of the currently executed thread, i.e. essentially the currently executed instruction in the active thread and all of its called functions. This allows to pinpoint the error injection location for later inspection.

The function *flip_single_bitvector_bit* is used in Line 39 as a helper function to inject a single bit error into an integer variable. It creates and constrains a symbolic integer value to non-deterministically select a single bit in the range defined by the *bitwidth* argument (Line 20-21). The global *bit* variable records the non-deterministic choice in Line 25 for later inspection (eventually the SMT solver will provide concrete values for non-deterministic choices). Based on the non-deterministic choice, the function performs the bit flip in Line 23 and returns the result. Please note, that the result itself becomes a symbolic expression.

C. Testbench

This section provides more details on the testbench focusing on assertion generation to guide the formal analysis. For illustration purpose, we discuss a (simplified) concrete example testbench for the IRQMP. Essentially, the input specifies incoming interrupts for the IRQMP and the output is a prioritized list of interrupt requests generated by the IRQMP.

When sending the interrupt mask $0b110$ as input and injecting a fault in the RTL model that results in wrong prioritization, the output $[2, 3]$ is observed instead of the expected output $[3, 2]$ - since higher interrupt lines have higher priority. Based on the input and faulty output the testbench is constructed. The monitoring logic records the observed interrupts in an array *irq*. Furthermore, it keeps track of the number of received interrupts in the *num_irqs* variable. Finally, the monitor asserts that *((irq[0] $\neq$ 2) || (irq[1] $\neq$ 3) || (num_irqs $\neq$ 2))* holds at the end of simulation. Essentially, it asserts that the observed output for the TLM model is not equal to the output of the faulty RTL model. Thus, the symbolic simulation engine will search for all possible error inject locations, that violate the assertion, i.e. produce the same failure at TLM as the faulty RTL model.

As an optimization, to prune irrelevant search paths which cannot produce the output of the faulty RTL model, we place *assume* instructions in the monitor. For this example, we would assume that the first received interrupt is 2 and the second is 3. Furthermore, we would assume that *num_irqs* < 2. Then a simple *assert (false);* can be placed at the end of simulation. Using stepwise assumptions during symbolic simulation, instead of a single assert in the end, can significantly reduce the considered search space, by pruning irrelevant search paths early.

VI. Case Study

We have evaluated our proposed fault correspondence analysis on the IRQMP model from the SoCRocket VP [11] as a case study. Our formal fault localization analysis is based on symbolic simulation approach of [12], [13]. In the following we report first experimental results for our proposed approach.

A. Experiments

All experiments were performed on an Intel 2.6 GHz machine with 16GB RAM running Linux. We employ Z3 v4.4.1 as our SMT solver.

For the experiments we use a set of representative test scenarios to cover different functionality of the IRQMP. Every scenario is using different inputs. Incoming interrupts are sent from the testbench to the IRQMP via register access (using a bus-transfer operation) or by writing to the *irq_in* wire. Furthermore, different priority levels are tested and resending of interrupt requests. Please note, all tests only use a single CPU and do not consider extended interrupts. The reason is that the RTL and TLM model are not functionally equivalent when using these features.

We use a set of representative fault injection locations for the RTL model. For every of these fault locations, a fault is injected in the RTL model for every test scenario. Table II shows a summary of our experimental results for injecting a bit flip during register configuration, interrupt prioritization computation as well as interrupt sending and acknowledgment. In particular, the following faults are injected:

1) a bit is flipped in the mask register, therefore the corresponding interrupt line is not processed or additional interrupt line activated;
2) a bit is flipped in the force register, therefore an additional interrupt needs to be processed or is omitted;
3) an incoming interrupt line is flipped, which has similar effect as the previous fault, but covers different functionality of the IRQMP;
4) a bit is flipped when resetting the incoming interrupt lines to zero;
5) the activation signal of the active interrupt acknowledgment wire is flipped, therefore an acknowledgment is missed;
6) a bit is flipped when sending the acknowledgment, this results in a wrong interrupt being acknowledged.
7) a bit is flipped when computing the interrupt prioritization, therefore a wrong interrupt is send or the order of two incoming interrupts is changed (e.g. flipping the second bit in 0b11 results in 0b01, thus interrupt line 1 is send first, even though line 2 has higher priority - since line 2 is still in the pending register, it will eventually be sent out too);
8) similar to the previous one, but at different location, where interrupts are interpreted as number instead of lines, e.g. 0b11 is interpreted as number 3 instead of interrupt lines 1 and 2;

Essentially, Table II combines the analysis results for all test scenarios for every fault injection location and reports the average values over all analysis runs. The first column shows a description of the RTL fault. The second column shows the type of the fault: *BT=Bus Transfer*, *WT=Wire Transfer* and *Comp=Computation*. The third and fourth column show the average runtime (in seconds), which is further divided in total analysis and SMT time. The SMT time shows how much of the analysis time is spent with solver queries. The number of solver queries is reported in the column *SMT Queries*. The

TABLE II
COMBINED RESULTS OF EXPERIMENTS (RUNTIMES IN SECONDS)

RTL Fault Location	Type	Average Runtime		Sym. Error	SMT	TLM Errors		
		TOTAL	SMT	Injections	Queries	Max	Min	Result
1) Mask Register Configuration	BT	285.86	184.18	162	30992	6	1	1
2) Force Register Configuration	BT	317.88	210.14	196	34690	7	4	4
3) Incoming Interrupt Wire	WT	317.80	207.18	166	35057	6	2	2
4) Incoming Interrupt Wire Reset	WT	690.74	454.03	311	74595	0	0	0
5) Missing Acknowledgement	WT	395.10	262.00	211	42839	7	6	6
6) Wrong Acknowledgement	WT	363.81	241.43	211	41502	1	1	1
7) Prioritization Logic 1	Comp	209.41	154.01	127	21856	12	4	4
8) Prioritization Logic 2	Comp	373.88	242.48	201	40538	9	0	0

column *Sym. Error Injections* shows how many errors have been injected during analysis of the TLM model. Please note that every error injection represents a non-deterministic bit flip. Finally, the column *TLM Errors* shows the maximum, minimum and result of error injection locations detected on the TLM model. The result denotes the number of TLM errors when computing the intersection of TLM errors for every test scenario. In other words the result column denotes the number of candidates for corresponding TLM errors found by our analysis.

For the RTL faults 4 and 8, no corresponding TLM errors are found (result column contains 0). The reason is that the TLM model is designed on a higher level of abstraction. For performance reasons callbacks are used that directly modify the internal model state without any delta cycles and context switches. Therefore, in case of RTL fault 4, a reset of the incoming interrupt lines is not necessary (and not available in the TLM model - setting the interrupt lines to a specific value is only applied once and not permanently until change). In the other case (RTL fault 8) it is due to a signal line where an error effect is delayed and propagated across a delta cycle. For both cases, it is possible that the RTL fault corresponds to multiple simultaneous TLM errors. For their localization, the symbolic error injection logic must be extended to support multiple errors. This extension is left for future work.

VII. CONCLUSION

In this paper we proposed an RTL-to-TLM fault correspondence analysis to improve the quality of error effect simulation using VPs. We employ formal methods to identify TLM error injection candidates for transient bit flips at RTL. First experiments on the IRQMP from the SoCRocket VP demonstrated the applicability and effectiveness of our approach in finding a small set of candidates for corresponding TLM errors.

REFERENCES

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

[2] 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.

[3] R. A. Shafik, P. Rosinger, and B. M. Al-Hashimi, "SystemC-based minimum intrusive fault injection technique with improved fault representation," in *IOLTS*, 2008, pp. 99–104.

[4] D. Lee and J. Na, "A novel simulation fault injection method for dependability analysis," *IEEE Design Test of Computers*, vol. 26, no. 6, pp. 50–61, Nov 2009.

[5] J. Perez, M. Azkarate-askasua, and A. Perez, "Codesign and simulated fault injection of safety-critical embedded systems using SystemC," in *EDCC*, 2010, pp. 221–229.

[6] A. Miele, "A fault-injection methodology for the system-level dependability analysis of multiprocessor embedded systems," *Microprocess. Microsyst.*, vol. 38, no. 6, pp. 567–580, Aug. 2014.

[7] M. L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *HPCA*, 2009, pp. 105–116.

[8] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *DAC*, 2013, pp. 1–10.

[9] A. Griesmayer, S. Staber, and R. Bloem, "Automated fault localization for C programs," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 4, pp. 95–111, 2007.

[10] H. M. Le, D. Große, and R. Drechsler, "Automatic TLM fault localization for SystemC," *TCAD*, vol. 31, no. 8, pp. 1249–1262, Aug. 2012.

[11] 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.

[12] 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.

[13] 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.

[14] G. Beltrame, C. Bolchini, and A. Miele, "Multi-level fault modeling for transaction-level specifications," in *GLSVLSI*, 2009, pp. 87–92.

[15] N. Bombieri, F. Fummi, and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *DATE*, 2008, pp. 396–401.

[16] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent systemc designs using mutation testing," in *HLDVT*, 2010, pp. 75–81.

[17] P. Lisherness and K.-T. (Tim) Cheng, "SCEMIT: A SystemC error and mutation injection tool," in *DAC*, 2010, pp. 228 –233.

[18] N. Bombieri, F. Fummi, and V. Guarnieri, "FAST: An RTL fault simulation framework based on RTL-to-TLM abstraction," *Journal of Electronic Testing*, vol. 28, no. 4, pp. 495–510, 2012.

[19] "GRLIB IP library," http://www.gaisler.com/index.php/products/ipcores/soclibrary.

[20] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.

[21] C.-N. Chou, C.-K. Chu, and C.-Y. R. Huang, "Conquering the scheduling alternative explosion problem of SystemC symbolic simulation," in *ICCAD*, 2013, pp. 685–690.

[22] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016.

[23] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.

[24] C. Flanagan and P. Godefroid, "Dynamic Partial-Order Reduction for model checking software," in *POPL*, 2005, pp. 110–121.