Towards Formal Verification of Real-World SystemC TLM Peripheral Models – A Case Study

Hoang M. Le[1] Vladimir Herdt[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
{hle,vherdt,dgrosse,drechsle}@informatik.uni-bremen.de

***Abstract*—SystemC-based *Virtual Prototypes* (VPs) serve as reference models for various activities in the modern design flow and therefore, the functional correctness of each individual components and the VPs as a whole should be subjected to rigorous formal verification. In the last few years, notable progress on SystemC formal verification has been made. This paper presents a case study on applying a recent approach to formally verify TLM peripheral models. To the best of our knowledge, this is the first formal verification case study targeting this important class of VP components. First, we show how to bridge the gap between the industry-accepted modeling pattern for TLM peripheral models and the semantics currently supported by SystemC formal verification approaches. Then, we report verification results for the interrupt controller of the LEON3-based SoCRocket VP used by the European Space Agency and reflect on our experiences and lessons learned in the process.**

I. Introduction

Nowadays the typical top-down *Electronic System Level* (ESL) [1] design flow starts with the creation of SystemC-based *Virtual Prototypes* (VPs). Essentially, a VP is a software simulation model of the entire hardware platform, created by composing *SystemC TLM* [2] models of the individual IP blocks (i.e. Instruction Set Simulators, bus and peripheral models, etc.). Serving as reference for (early) embedded software development and hardware verification, the functional correctness of the VPs is more important than ever and hence subjected to rigorous formal verification.

In the recent years, notable progress on SystemC formal verification has been made. From the first approaches (about 10 years back), which only supported RTL-like models, a new generation of SystemC verifiers [3], [4], [5], [6], [7], [8], [9] has emerged to cope with the abstract modeling styles of SystemC TLM. In this context, an extensive set of academic SystemC benchmarks is available. From the practical perspective however, it is mandatory to investigate the applicability of existing formal approaches on real-world VPs.

A first attempt has been made in [10], where the successful application of [5] on a simplified ARM AHB TLM-2.0 model is reported. Besides such memory-mapped buses, that are modeled using the TLM-2.0 standard [2], TLM peripherals (i.e. TLM models of on-chip peripherals such as timer, interrupt controller, memory controller, ADC, etc.) occupy an essential portion of real-world VPs. For this class of models, the separation of communication, behavior and storage in a reusable way has also come into sharp focus [11]. This resulted in specialized modeling concepts for memory-mapped register access and abstract wire communication (for interrupts and reset) at TLM. These concepts have been widely embraced and implemented in several industrial solutions, e.g. the *SystemC Modeling Library* (SCML) by Synopsys [12] or the joint proposal of TLM extensions by Cadence and ST [13], as well as in the open-source offer GreenReg [14], which is essentially a refinement of Intel's *Device Register Framework* (DRF).

In this work, we report preliminary results on formal verification of TLM peripheral models, which is part of a long-term effort towards a formally verified VP. The main contribution is a feasibility case study involving a real-world TLM model of an interrupt controller. The model is taken from the recently available open-source VP SoCRocket [15], which represents the popular LEON3 SoC platform [16] and has been developed in a joint effort with the *European Space Agency* (ESA). In order to achieve the results, a new modeling layer has been created to handle the above mentioned modeling constructs, i.e. in particular register and wire modeling at TLM. The modeling layer and the models are available at www.systemc-verification.org/sissi, where we will also incrementally add more extracted models from SoCRocket to enrich the set of benchmarks available to the research community.

II. TLM Peripheral Modeling in SystemC

This section gives a compact overview on two industry-accepted modeling concepts for TLM peripherals: register and wire modeling. We use the particular implementation within SocRocket, which implements GreenReg-like API on top of the above mentioned extensions by Cadence and ST, as demonstration.

A. TLM Register Modeling

In practice, for TLM peripherals, memory-mapped (control) register access is very common. Hence, the design productivity can be improved by greatly simplifying the description of registers and providing a reusable API for typical register features. Now consider Fig. 1 as an example. First, the member function *create_register* of the register file class is called to create a new register in Line 2. The arguments are the short register name, the full description (Line 2-3), the address offset

This work was supported in part the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E, 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  void Irqmp::init_registers() {
2    regfile.create_register("pending", // register name
3      "Interrupt_Pending_Register", // description
4      0x04, // address offset
5      0x00000000, // initial value
6      IR_PENDING_EIP | IR_PENDING_IP) // write mask
7      .callback(POST_WRITE, this, &Irqmp::pending_write);
8    // ...
9  }
```

Fig. 1. TLM Register Modeling Example

(Line 4), the initial value (Line 5) and the write mask (Line 6). The address offset allows to map read/write TLM-2.0 bus transaction to the register. The write mask, represented by the ORed value of two integer constants, determines which bits are allowed to be written by such a transaction.

After its creation, behavior is attached to the register by adding *function callback* (Line 7). Whenever the register is accessed via a TLM transaction, the callback is invoked. The invocation can be before/after reading/writing. In the example, the function *pending_write* of the *Irqmp* class will be called after the register value is updated (i.e. *POST_WRITE*). Furthermore, in a similar way, callbacks can also be registered on value changes of a single bit or a group of bits instead of the whole register. For the simplicity of presentation, we only show the latter.

B. TLM Wire Modeling

A major interoperability problem occurs in practice, since the TLM-2.0 standard does not specify how TLM wires are modeled. TLM wires are needed in particular for interrupt and reset signals where standard transaction payloads are too costly from both modeling perspective and simulation performance. With the naive solution of using *sc_signal* of SystemC, value updates require context switches in SystemC kernel. In the proposal by Cadence and ST (also in discussion for standardization within Accellera), value updates to TLM wires are immediately available (thus sidestepping the SystemC kernel) and a callback can be registered to execute desired behavior on value changes in a similar manner to TLM register callbacks.

III. Bridging the Modeling Gap

An analysis of the existing SystemC formal verification approaches reveals that none of them directly supports the described TLM peripheral modeling concepts. Developing a capable SystemC frontend, which additionally handles the TLM register and wire modeling constructs, would be very daunting. The other alternative is to map these constructs into intermediate representations supported by existing verification approaches. These representations, such as the threaded C subset supported by [7] or the *Intermediate Verification Language* (IVL) proposed in [8], however, contain only primitive imperative programming constructs. A manual translation of the abstract peripheral modeling constructs into these would require considerable amount of very error-prone work.

On the other hand, TLM peripheral modeling constructs are implemented as C++ classes and thus, they can be naturally modeled using *object-oriented* (OO) constructs. Therefore,

```
1  struct RegCallback {
2    abstract void call(RegCallback *this);
3  }
4  #define PRE_WRITE 0
5  ...
6  struct Register {
7    uint32_t value;
8    uint32_t write_mask;
9    RegCallback *callbacks[MAX_CALLBACKS];
10   void write(Register *p, uint32_t value) {
11     p->value = value;
12   }
13   void bus_write(Register *p, uint32_t value) {
14     if (p->callbacks[PRE_WRITE] != nullptr)
15       ::call(p->callbacks[PRE_WRITE]);
16     ::write(p, value & p->write_mask);
17     if (p->callbacks[POST_WRITE] != nullptr)
18       ::call(p->callbacks[POST_WRITE]);
19   }
20   // ...set_bit, get_bit, read, bus_read...
21   void add_callback(Register *p, uint32_t index,
22         RegCallback *c) {
23     p->callbacks[index] = c;
24   }
25 }
```

Fig. 2. Register Class in XIVL

```
1  struct PendingWriteCallback extends RegCallback {
2    Irqmp *receiver;
3    virtual void call(PendingWriteCallback *this) {
4      ::pending_write(this->receiver);
5    }
6  }
7  RegCallback *create_pending_write_callback(Irqmp *p) {
8    PendingWriteCallback *c = new PendingWriteCallback;
9    c->receiver = p;
10   return c;
11 }
```

Fig. 3. Example Callback Implementation in XIVL

our solution is to extend the open-source available IVL with minimal OO support, just enough to capture the behavior of TLM registers, wires and callbacks. To be more specific, we have added support for classes, inheritance, virtual methods with overrides and dynamic dispatch. We refer to this extended version of IVL as XIVL in the remainder of this paper.

Each TLM register or wire is modeled as an XIVL class with various member variables such as its values, associated callbacks, etc. A generic callback is modeled as an IVL class with an abstract member function, which should be implemented by a specific callback subclass to add the desired behavior. The register file of a TLM peripheral is modeled as an array of registers. In the following, we discuss the modeling of TLM registers and callbacks in XIVL in more details. The modeling of TLM wires in XIVL is analogous and thus omitted for a more compact presentation.

An XIVL implementation of the generic *Register* class is shown in Fig. 2. The *bus_read/write* functions mimic a TLM read/write transaction to the register, since they will additionally invoke registered callbacks as shown in Line 15 and Line 18, respectively. The *add_callback* function shown in Line 21 is used to register a generic callback object that adheres to the *RegCallback* interface shown in Line 1. A concrete callback implementation is shown in Fig. 3. On construction, the receiver class is stored as a member of

```
1  void init_registers(Irqmp *p) {
2    RegCallback *c = create_pending_write_callback(p);
3    p->regfile[0x04] = create_register( // 0x04=address off.
4      0x00000000, // initial value
5      IR_PENDING_EIP | IR_PENDING_IP); // write mask
6    ::add_callback(p->r[0x04], POST_WRITE, c);
7    // ...
8  }
```

Fig. 4. Register Initialization in XIVL

the *PendingWriteCallback* class, as shown in the construction function in Line 9. This allows to delegate the callback to the corresponding function of the receiver in Line 4, in this case the *pending_write* function of the *Irqmp* class. Basically, the code fragment in Fig. 3 is a template that is replicated, while adapting types and arguments, for every required callback. Finally, Fig. 4 shows an XIVL fragment, which corresponds to the SystemC code in Fig. 1, to create and initialize a register. The *create_register* function (implementation not shown in Fig. 2) allocates and initializes a new register with the given initial value and write mask. The resulting register is stored in the register file array *regfile*, which is a member of the *Irqmp* class. The initial value and write mask are associated with the register, whereas the address offset corresponds to the array index in *regfile*.

On a final note, XIVL mostly resembles C++, except for some following minor differences: 1) the *extends* keyword is used for inheritance; 2) the implicit first argument *this* to member functions is declared and passed explicitly; 3) member functions are called by prepending *::* before the function name, which allows to distinguish between static and potentially dynamic calls.

IV. Case Study

This section presents our effort to formally verify the TLM model of the Interrupt Controller for Multiple Processors (IRQMP) of the LEON3-based VP SoCRocket [15]. Thanks to the added modeling layer, we were able to translate the original SystemC TLM model into XIVL rather effortlessly. In the following first the basics of the IRQMP are described. Then, the obtained verification results are reported.

A. Interrupt Controller for Multiple Processors

The IRQMP is an interrupt controller supporting up to 16 processors. Its functionality is to process incoming interrupts by applying masking and prioritization of interrupts for every processor. Fig. 5 shows an overview of the TLM model. It consists of a register file, several input and output wires and an APB Slave interface. The register file and wires are implemented as described in Section II. The SystemC thread *launch_irq* and the callbacks implement the behavior logic of the IRQMP. The callbacks can update register values and activate the *launch_irq* thread, which is responsible to compute the interrupt requests for each processor.

In the following, we briefly summarize the specified behavior of the IRQMP described in the specification from [16]. IRQMP supports incoming interrupts (*irq_in*) numbered from 1 to 31 (interrupt line 0 is reserved/unused). Lines 15:1 are

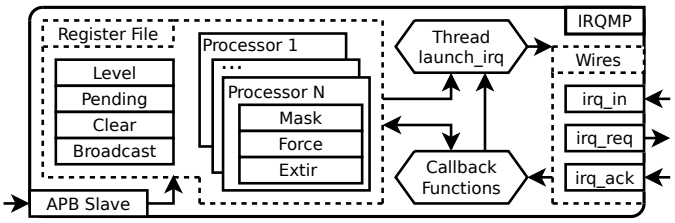

Fig. 5. IRQMP Overview

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 register file contains shared and processor-specific registers. Every register has a bit width of 32. Each bit naturally represents an interrupt line.

1) Interrupt Handling: IRQMP computes interrupt requests for every processor P by first, combining Pending with Force and Mask register of P using bitwise operations as ((Pending | P.Force) & P.Mask) and then, select the highest prioritized interrupt. 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.

Handling of incoming interrupts depends on the Broadcast register. A high bit in Broadcast means that the corresponding interrupt should be broadcasted. A broadcasted interrupt is written to the Force register of all processors, therefore it also has to be acknowledged by every processor. Otherwise an incoming interrupt is written to the shared Pending register. In this case, it is sufficient when one processor acknowledges the interrupt. A processor P can acknowledge an interrupt by writing to the *irq_ack* wire. The interrupt will be cleared from Pending as well as from P.Force. A bit in Pending can also be explicitly cleared by raising the corresponding bit in Clear.

2) Extended Interrupts: IRQMP has only 15 outgoing interrupt lines (*irq_req*) for every processor. Therefore, all extended interrupts are mapped onto a single regular interrupt, determined by the value of $EIRQ \in \{1..15\}$ (with $EIRQ = 0$ indicating regular mode). Masking and prioritization of incoming extended interrupts is handled similarly to regular interrupts. When the interrupt $EIRQ$ is acknowledged by process P, in addition to clearing the Pending register, the line number of the extended interrupt is written into P.Extir, which allows to fetch the last acknowledged extended interrupt.

B. Formal Verification

For formal verification, we employ a re-implementation of the path-based symbolic simulation approach for SystemC proposed in [8] supporting XIVL. Due to the absence of a TLM property language (and a tool for property instrumentation), we manually build a formal verification environment. It consists of a symbolic input driver to IRQMP and a checker to monitor its responses, respectively. IRQMP is treated as blackbox during the verification. Our evaluation results consist of two parts.

TABLE I
VERIFICATION RESULTS

Verification scenarios		Verdict	Time (sec.)
S1	Broadcast		
	S1.1 Sent to all CPUs	Safe	56.67
S2	Masking		
	S2.1 Single CPU with force	Unsafe	15.01
	S2.2 Two CPUs no force	Safe	24.15
	S2.3 Extended interrupts	Unsafe	24.84
S3	Priorization		
	S3.1 Single request	Safe	13.79
	S3.2 With extended interrupts	Safe	17.09
	S3.3 Acknowledgement roundtrip	Safe	2093.42
	S3.4 With mask	Safe	3766.86
	S3.5 With mask and two CPUs	-	T.O.
	S3.6 With extended interrupts	Unsafe	306.48

1) Tackling Path Explosion in Symbolic Simulation: Using a symbolic driver to make all inputs symbolic allows to explore all possible behaviors of IRQMP. However, in preliminary experiments, we ran into the path explosion problem in path-based symbolic simulation, as the number of feasible exploration paths grows exponentially. To mitigate this problem, we have implemented a path merging algorithm. Generally, merging results in more complex solver queries, while explicit exploration suffers from path explosion problem. Our merging strategy is based on manual annotations of branches and loops, which allow a fine grained selection between path merging and explicit path exploration. Additionally, we restricted the number of symbolic inputs based on the functionality under verification. We apply a number of different verification scenarios focusing on different aspects of IRQMP as described in the next section.

2) Verification Scenarios and Results: Our symbolic verification scenarios are partitioned into three groups based on the functionality under verification: Broadcasting, Masking and Prioritization of interrupts. Table I summarizes the verification results. The table shows the scenario ID, e.g. S1.1, along with a small description in the first column. The second column (*Verdict*) shows the verification result, where *Unsafe* indicates a bug in IRQMP and *Safe* indicates a successful verification. The last column shows the verification time in seconds. All experiments have been performed on an AMD 3.4 GHz machine running Linux. The time and memory limit has been set to 2 hours and 6 GB, respectively.

The first group checks broadcast functionality. It contains a single scenario that verifies that IRQMP sends a broadcasted interrupt to every processor. For this scenario, all 16 processors are used. The symbolic input driver selects non-deterministically the interrupt to be broadcasted (by setting the same bit in *irq_in* and Broadcast).

The Masking group verifies that every observed interrupt request for a processor P is not masked for P. The symbolic input driver in all S2 scenarios makes all Mask registers and *irq_in* symbolic. S2.1 fails because the Force register is erroneously applied after the Mask register during interrupt request computation in IRQMP. S2.3 uses two processors and extended interrupts (i.e. $EIRQ$ is additionally symbolic). It has revealed a subtle error that requires at least two processors: an $EIRQ$ interrupt can be sent to a processor even though all extended interrupts are masked. Non-formal verification techniques could easily miss this bug.

The Prioritization group checks whether interrupt requests come in ordered according to their priorities for every processor. In all scenarios of this group, the Level register and *irq_in* are set to be symbolic. The scenarios are constructed in increasing complexity order. S3.1 uses only one processor and no requested interrupts are acknowledged. S3.2 is similar but allows extended interrupts additionally. In the remaining four scenarios, all interrupts are acknowledged via either the *irq_ack* wires or the Clear register. Furthermore, S3.4 adds a symbolic Mask, S3.5 a second processor and S3.6 extended interrupts. S3.6 detects an error in handling extended interrupts: extended interrupts are not automatically cleared when the $EIRQ$ interrupt is acknowledged, therefore IRQMP keeps requesting extended interrupts without terminating.

C. Summary

We were able to prove important functional properties of IRQMP as well as to detect three major bugs. This clearly demonstrates the feasibility of formal verification of TLM peripherals. On the other hand, the case study unveils open challenges to be addressed in future work, including e.g.

- Incorporating a SystemC TLM property language into the formal verification flow to enable the automatic generation of symbolic input driver and checker;
- Additional language features to further close the gap between SystemC TLM and the current XIVL: similar to the added OO features this would allow to translate a larger set of SystemC programs without complex error-prone manual transformations;
- An intelligent path merging algorithm for symbolic simulation, which can automatically find a good balance between merging and explicit exploration, to replace the current one, which requires manual annotations.

REFERENCES

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.
[2] *IEEE Standard SystemC LRM*, IEEE Std. 1666, 2011.
[3] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," *TODAES*, vol. 15, no. 3, pp. 21:1–21:32, Jun. 2010.
[4] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.
[5] P. Herber, M. Pockrandt, and S. Glesner, "Transforming SystemC transaction level models into UPPAAL timed automata," in *MEMOCODE*, 2011, pp. 161–170.
[6] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. R. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.
[7] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *TCAD*, vol. 32, no. 5, pp. 774–787, 2013.
[8] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *DAC*, 2013, pp. 116:1–6.
[9] V. Herdt, H. M. Le, and R. Drechsler, "Verifying systemc using stateful symbolic simulation," in *DAC*, 2015, pp. 49:1–49:6.
[10] M. Pockrandt, P. Herber, and S. Glesner, "Model checking a SystemC/TLM design of the AMBA AHB protocol," in *ESTIMedia*, 2011, pp. 66–75.
[11] T. Kogel, "Peripheral modeling for platform driven ESL design," in *Platform Based Design at the Electronic System Level*. Springer Netherlands, 2006, pp. 71–85.
[12] Synopsys Inc., "SystemC Modeling Library (SCML)," https://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi.
[13] S. Swan and J. Cornet, "Beyond TLM 2.0: New Virtual Platform Standards Proposals from ST and Cadence," 2012, presented at NASCUG at DAC.
[14] "GreenReg," http://www.greensocs.com/projects/GreenReg.
[15] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "Socrocket - A virtual platform for the European Space Agency's SoC development," in *ReCoSoC*, 2014, pp. 1–7, available at http://github.com/socrocket.
[16] "GRLIB IP library," http://www.gaisler.com/index.php/products/ipcores/soclibrary.