Process algebra as a common framework for hardware/software coverification

Citation for published version (APA):

Raffelsieper, M., Mousavi, M. R., & Sleuters, J. (2011). Process algebra as a common framework for hardware/software coverification. *IEEE Embedded Systems Letters*, 3(1), 9-12.
<https://doi.org/10.1109/LES.2010.2091621>

DOI:

[10.1109/LES.2010.2091621](https://doi.org/10.1109/LES.2010.2091621)

Document status and date:

Published: 01/01/2011

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Process Algebra as a Common Framework for Hardware/Software Coverification

Matthias Raffelsieper, *Student Member, IEEE*, Mohammad Reza Mousavi, and Jack Sleuters

Abstract—This letter presents the practical issues concerning late and insufficient verification of low-level software on hardware platforms developed by our industrial partner. To overcome these issues, we propose a coverification platform based on process algebra. The descriptions of hardware and software, and their interface are translated into a common process-algebraic platform, and formal verification techniques are used to check the conformance of the two descriptions. We present the results of our first attempt towards this goal, discuss the lessons learned, and present the road-map for future research.

Index Terms—Formal verification, low-level software, process algebra, software/hardware coverification.

I. INTRODUCTION

DURING the ongoing cooperation with our industrial partner, Virage Logic, it appeared that the developers of low-level software are currently faced with a major obstacle in the verification phase: they may only start the execution of tests after the development of the actual hardware. Since projects often run behind the schedule, the verification of low-level software coincides with a tight deadline to deliver the product and thus, results in two undesired consequences: due to the short time available for testing, first, the system may be insufficiently tested, and second, if a fault is detected there is little time available to isolate its cause and remove it. The problem is intensified if a serious flaw is traced back to the developed hardware; in such a case a respin may be in order (if the problem cannot be masked by the low-level software) and this endangers the planning and the financial success of the project.

Moreover, there is a gap between the view of the environment on the software side and the hardware side. For software, the interaction with the hardware is done by memory-mapped registers, whereas the hardware sees some kind of protocol that is to be followed.

One emerging trend that may be seen as an opportunity is the ever-growing trend of developing high-level descriptions of both hardware and software systems. This trend started off with hardware descriptions provided in languages such as VHDL and Verilog; moreover, we currently witness the rise of abstraction level to system- and transaction-level in languages such as

(System)Verilog and SystemC. These high-level models may be used as the virtual platform for testing and verifying the software on top of them.

Motivated by the former problem and the latter opportunity, we propose an approach for software/hardware coverification: we propose to transform both software and hardware descriptions into a formal language, a process algebra called mCRL2 [4] in our case, and perform formal verification in the common framework. This framework allows the developer to verify the low-level software against the hardware specification rather than testing it on the manufactured chip, enabling early verification of low-level software. Hence, the main contribution of this letter is to put forward this methodology and provide a proof of concept for it by applying it to a case study provided by Virage Logic. We mechanize some of the steps involved in this methodology in order to facilitate our ongoing experiments with more cases. We report about our initial findings and the envisaged roadmap.

Related Work: In [8], it is reported that more and more companies are developing new designs at the system level. Furthermore, it is observed that even when hardware and software are correct, there might be problems when combining them in a system-on-chip. Verification is usually done through simulation, i.e., only a limited set of behaviors is checked. This differs from our approach, as we use formal techniques to check that *all* possible behaviors do not violate a given property. Formal techniques are also used in [7], where rewriting logic is used to verify properties of hardware descriptions given in the ABEL language on which assembly programs run. However, this approach cannot deal with peripheral devices in different clock domains, since it assumes a single global (and implicit) clock. Furthermore, the approach is limited to trace checking, which is restricted to linear time properties for finite traces and can only verify systems that are completely deterministic. A similar approach is presented in [9], where coverification of Verilog hardware descriptions and xUML software descriptions is done using a model checker. The authors also observed a mismatch between the hardware and the software semantics. They solve this by introducing what they call a *bridge*, which translates between the two views. However, in contrast to our work, their bridge depends both on the hardware and the software. In our work, we use a *transactor*, that only depends on the communication between hardware and software, but otherwise is independent of the specific hardware and software used. This way, we can reuse a transactor in other verifications where hardware and software communicate in the same way.

The rest of this letter is organized as follows. In Section II, we present our proposed methodology. This way, we do not have to recreate the whole model when just a part changes, but instead only the model of the modified part has to be adjusted.

Manuscript received October 01, 2010; accepted November 04, 2010. Date of publication November 11, 2010; date of current version March 23, 2011. This manuscript was recommended for publication by S. K. Shukla.

M. Raffelsieper and M. R. Mousavi are with the Department of Computer Science, Eindhoven University of Technology, Eindhoven, 5600 MB, The Netherlands (e-mail: M.Raffelsieper@tue.nl; M.R.Mousavi@tue.nl).

J. Sleuters is with TASS Software Professionals, Eindhoven, 5616 VB, The Netherlands (e-mail: jack.sleuters@viragelogic.com).

Digital Object Identifier 10.1109/LES.2010.2091621

Fig. 1. Schematic overview of our approach.

Section III then presents a first case study we performed for a general purpose input–output interface (GPIO). In Section IV, we report on some of the lessons learned through the presented case study and other ongoing experiments. Finally, Section V draws conclusions and discusses future topics to be investigated.

II. PROPOSED METHODOLOGY

A schematic overview of our approach is depicted in Fig. 1. To solve the problem of verifying low-level software that is written for a specific hardware, we create a combined model of both components for which we can then check system-wide properties. To enable as much reuse as possible, we build the combined model from individual models for the hardware, the low-level software, and the used bus connection. This way, one only has to update a model when a change in that part has been made.

To this end, we give a mechanized compositional translation from the hardware description into the formal domain. Such a description can be defined at the system level to allow development and verification of the software in parallel with the development of the detailed hardware description (e.g., by using our mechanized translation from SystemC into the process algebra mCRL2 reported in [5]). However, it is also possible to use RTL descriptions, as in our example described below. This results in a model in the formal language mCRL2 [4], where for each processes in the VHDL description a parallel process in mCRL2 is created.

Apart from translating the constructs in the hardware description, the semantics of the description language is formally described in terms of a number of fixed processes in mCRL2. In our case, we add two parallel mCRL2 processes that take care of administrating the signals and scheduling the enabled processes. The first process, called *RegisterFile*, stores for every signal its current and new value. This is done on the one hand to be able to detect edges in a signal and on the other hand to execute all hardware processes with the same set of input values, as it is required by the VHDL execution semantics described in the VHDL standard [6]. In the second process, called *Kernel*, the execution semantics is implemented. This works by first updating the values of the signals and then reevaluating those processes that are sensitive to one of the changed signals. This is repeated until no process is activated anymore. In such a state, new input values are accepted, which might again trigger

the above cycle, called a *delta cycle*. We do not impose any restriction on the input values sent to the hardware. Thereby, we allow different clocks to be running at different speeds and even allow them not to be synchronized at all. Furthermore, we allow the Kernel to schedule hardware processes in any order. This enables nondeterministic evaluation of hardware, instead of imposing some arbitrary order, as usually done by simulators. Thus, our approach can also find bugs due to such nondeterminism, which would never be found by a simulator.

Note that the semantics described above is specific to the language VHDL used in our case study. However, it can easily be adopted to other language semantics (e.g., see [5] for the system-level language SystemC, which is also based on delta cycles).

An mCRL2 model is also created from the low-level software written in C. This model has an interface to the memory abstraction, which provides read and write actions. The write action has as arguments both the address and the value to be written. The read action however is split into two parts, one sending the address to be read from, and the other receiving the value that was read from that address.

To combine the two models, one needs to translate between the memory view of the software and the view of the hardware, which is usually some bus interface. For this purpose, we introduce a bus-protocol-specific *transactor*, that does exactly this translation. Such a transactor is specific for the bus protocol, but neither depends on the software nor on the hardware. Thus, such a transactor has to be created only once for each bus protocol and can be reused afterwards for any design using the specific protocol. For our case study, we created such a transactor for the industry standard APB bus [1].

Finally, we put all three models, i.e., the model of the hardware, that of the software, and the transactor model, together into one system model and abstract from the internal (unobservable) interactions within each model. This system model represents the external (interface-level) behavior of the software on a CPU that is connected to the hardware by the specific bus. For this model, one can then define properties that describe the expected behavior and verify them using the model-checking tools from the mCRL2 toolset [3] and the LTSMIN toolset [2].

III. CASE STUDY

We have applied the approach described in the previous section to a GPIO (having output pins that can be driven either high or low and input pins that can read either high or low values). This GPIO consists of both an RTL description in VHDL of the hardware and a low-level driver written in C. The hardware interacts with its environment by means of the APB bus, whereas the driver assumes that the registers of the hardware are mapped into the memory of the processor it is running on.

We have developed a prototype tool to translate the model of the hardware in order to facilitate this case study as well as other ongoing experiments. We also created a model of the driver software manually. Finally, we created a (reusable) model for an APB transactor.

In contrast to the hardware, the software does not contain an explicit clock, instead for the software it is only relevant that the statements are executed in the given order. The software runs in isolation on the CPU, until it eventually reaches a statement

Fig. 2. Message flow for reading using the APB transactor.

```

apb_do_en_proc: PROCESS
  (apb_enable, apb_sel, apb_write)
BEGIN
  IF apb_sel = '1' AND apb_write = '0'
  THEN
    apb_do_en <= apb_enable;
  ELSE
    apb_do_en <= '0';
  END IF;
END PROCESS;

```

Fig. 3. RTL example code.

that communicates with the hardware via the registers mapped into memory. At these points, the software blocks and waits for the communication to complete. The communication request is received by the transactor, which then follows the APB bus protocol to forward the request to the hardware. This protocol references the bus clock, so the transactor has to interact with it. Finally, the hardware notifies the transactor when the request has been processed, together with any values that were requested (such as for example read values). Then the control (and possibly requested values) are passed back to the software, which is then allowed to continue its execution.

As an example, we show the flow of execution for reading the value of a register in Fig. 2. The first step consists of the driver sending a read request to the transactor. It then follows, in the second step, the APB protocol to issue a read request to the hardware. This takes a number of exchanges between the transactor and the hardware, as suggested in the figure. In the third step, the hardware computes the value to be returned, which is thereafter obtained by the still running APB protocol for reading in the fourth step. Finally, in the fifth step, the value to be read is returned to the software and it can continue executing.

To give an intuition about the translation of the hardware, we show in Fig. 3 the VHDL code of the hardware process that is responsible for computing the signal `apb_do_en`, which determines whether read data is to be provided on the APB bus or not. As required by the VHDL semantics, this process is enabled whenever one of the signals `apb_enable`, `apb_sel`, or `apb_write` changes its value, indicated in the parameters of the process. Based on these values, the new value of `apb_do_en` is then computed.

```

proc proc_apb_do_en_proc =
  sum events: List(RegName).
  rcvStartProc(apb_do_en_proc, events).
  (
    sum val_apb_sel: List(Bool).
    rcvReadReg(apb_sel, val_apb_sel).
    sum val_apb_write: List(Bool).
    rcvReadReg(apb_write, val_apb_write).
    ( (val_apb_sel == bool2bits(true))
      &&
      (val_apb_write == bool2bits(false))
    )
  ) ->
  sum val_apb_enable: List(Bool).
  rcvReadReg(apb_enable, val_apb_enable).
  sndWriteReg(apb_do_en, val_apb_enable)
  <>
  sndWriteReg(apb_do_en, bool2bits(false))
  ).
  sndFinishedProc(apb_do_en_proc).
  proc_apb_do_en_proc;

```

Fig. 4. mCRL2 example code.

In the mCRL2 translation of this hardware process, which is shown in Fig. 4, the first action is called `rcvStartProc(apb_do_en_proc, events)`. This action is issued by the kernel process whenever one of the above-mentioned three variables changes its value. The first argument of this action, `apb_do_en_proc`, indicates that it is *this* process that shall be started, as the action is used for all processes in the system. The second argument, `events`, contains a list of variables that have actually changed their value. This allows to detect, for example, positive edges. After having received the start signal, the process first reads the values of the variables `apb_sel` and `apb_write`, using the action `rcvReadReg`. These values have to be made explicit since the values are not stored in this process, but instead in the process `RegisterFile` mentioned above and ensures that all variables are shared among the processes. Using the values of the variables, which are prefixed with `val_`, the IF statement in the VHDL source can be evaluated. Such a conditional statement is also available in mCRL2 with a change of syntax: a VHDL expression of the form `IF b THEN  $s_1$  ELSE  $s_2$  END IF` becomes `b ->  $s_1$  <>  $s_2$` in mCRL2. This can also be seen in the translated mCRL2 code. There, depending on the value of the condition, the variable `apb_do_en` is assigned a new value. This again has to be done by communicating with the `RegisterFile` process, using the action `sndWriteReg`. Finally, the action `sndFinishedProc(apb_do_en_proc)` signals to the kernel that this process has finished executing, so that the kernel can schedule the next process. Furthermore, the process has to be reset into a state where it again can be started, so in the last line of the translated mCRL2 code it recursively calls itself, where it will wait for the kernel to start it again.

Figs. 5 and 6 show the C code and the mCRL2 code for a function taken from the driver, respectively. The function shown implements reading a value from a pin. In mCRL2, we create a process definition for each line of the driver, for better readability and modularity; this way, we can call lines of code from different locations by using the corresponding process definition. This is for example desirable for translating loops

```

uint32_t
PinsRead(int gpioUnitId, uint32_t* pPinData) {
    uint32_t regs;
    regs = cfg_Gpio_Base[gpioUnitId]
        + GPIO_READ_REG_OFFSET;
    GPIO_GEN_READ(regs, *pPinData);
    return 0;
}

```

Fig. 5. Example C driver code.

```

proc PinsRead =
    sum gpioUnitId : Nat.
        rcv_PinsRead(gpioUnitId).
        PinsRead_body_1(gpioUnitId);
proc PinsRead_body_1(gpioUnitId: Nat) =
    tau.
        PinsRead_body_2(gpioUnitId,
            valueAtIndex(cfg_Gpio_Base, gpioUnitId)
            + GPIO_READ_REG_OFFSET
        );
proc PinsRead_body_2
    (gpioUnitId: Nat, regs: Nat) =
    sndReadMemReq(regs).
    sum v: Nat.rcvReadMemResp(v).
    PinsRead_body_3(gpioUnitId, regs, v);
proc PinsRead_body_3
    (gpioUnitId: Nat, regs: Nat, pinData: Nat) =
    snd_return_PinsRead(0, pinData).
    PinsRead;

```

Fig. 6. mCRL2 code for the PinsRead function.

into mCRL2. The translation of the driver function waits in the main process `PinsRead` for a call from an application by means of the action `rcv_PinsRead(gpioUnitId)`. Note that we currently do not support pointers, instead we will move the value returned in `pPinData` into the return value. If the function is called, process `PinsRead_body_1` computes the address to read from, using the function `valueAtIndex` to access the array and the addition operation. This address value is used in the process `PinsRead_body_2` to send a read request to the transactor, by means of the action `sndReadMemReq`. The transactor will eventually return the read value, which is received by the action `rcvReadMemResp`. This value will then, as already mentioned above, be returned in the action `snd_return_PinsRead` together with the return code. Finally, the function is made to wait for another call by recursively calling the initial process `PinsRead`.

The translated models for the hardware and the software, together with the model of the transactor, were then combined into a complete system model. This system model was approximately 1200 lines of mCRL2 code. As an example property, we successfully verified that the resulting model has no deadlock, i.e., it never reaches a state in which no further actions are possible.

IV. LESSONS LEARNED

Our case study has provided a proof of concept for the general methodology of using process algebra and in particular mCRL2 as a common platform for hardware/software coverification. There are a number of features in mCRL2 and its associated toolsets which facilitated this case study. First, mCRL2

provides an extensible language for expressing abstract data types in combination with behavioral specification. This feature is absent in many other pure process algebraic frameworks. Second, there is both support for explicit as well as symbolic-state analysis of the state-space. Although for practical verification purposes, we were forced to use the symbolic-state analysis, the explicit-state tools allow us to visualize the (partial) state-space and examine its structure in order to devise effective reduction techniques and improve the structure of the translation.

During our experiments with the translation and verification, we gathered the following observations. First, an effective and reusable technique in our methodology is to interface low-level software and hardware using a transactor model. This way, we also allow the software model to run infinitely fast (by a series of instantaneous actions) in between the interface calls. Second, there is a huge branching structure in the state-spaces involving hardware specifications with multiple clock domains, however, most of these branches turn out to be confluent. Finding syntactic (static) criteria to identify these confluent structures lead to an effective use of the confluence reduction technique on such specifications. Third, we gained some reduction in the state space of the system by inlining combinational logic in the VHDL processes of sequential gates surrounding it.

V. CONCLUSION

We presented a methodology that, given a combination of hardware and software together with a specified bus interconnection, can be used to model-check properties of a complete hardware/software system. We have applied this methodology to a small example and successfully verified deadlock freedom.

In the future, we plan to extend our approach by the treatment of more sophisticated hardware designs. These case studies will provide more insight for the to-be-developed abstraction and reduction techniques for large system-level designs. Finally, a parameterized verification technique is required to verify software and hardware with multiple timing domains and obtain bounds on the relative timing specifications of the two domains to guarantee their correct behavior.

REFERENCES

- [1] ARM, AMBA 3 APB Protocol Specification v1.0 2004.
- [2] S. C. C. Blom, J. C. van de Pol, and M. Weber, "LTSMIN: Distributed and symbolic reachability," *Proc. CAV*, pp. 354–359, 2010.
- [3] J. F. Groote *et al.*, "The mCRL2 toolset," *Proc. of WASDETT*, 2008.
- [4] J. F. Groote *et al.*, "The formal specification language mCRL2," in *Dagstuhl Seminar Proc.*, 2007, vol. 6351.
- [5] H. Hojjat, M. R. Mousavi, and M. Sirjani, "Process algebraic verification of SystemC codes," *Proc. ACSD*, pp. 62–67, 2008.
- [6] *VHDL Language Reference Manual*, IEEE Standard: 1076-2008, 2009.
- [7] M. Katelman and J. Meseguer, "A rewriting semantics for ABEL with applications to hardware/software co-design and analysis," *ENTCS*, vol. 176, pp. 47–60, 2007.
- [8] S. Swan, "SystemC transaction level models and RTL verification," in *Proc. Design Autom. Conf. (DAC)*, San Francisco, CA, 2006, pp. 90–92.
- [9] F. Xie, X. Song, H. Chung, and R. Nandi, "Translation-based coverification," in *Proc. MEMOCODE*, Verona, Italy, 2005, pp. 111–120.