# Safe IP Integration Using Container Modules

Rolf Drechsler

*Group for Computer Architecture*
*University of Bremen / DFKI*
*Bremen, Germany*
*Email: drechsler@uni-bremen.de*

Ulrich Kühne

*Group for Computer Architecture*
*University of Bremen*
*Bremen, Germany*
*Email: ulrichk@cs.uni-bremen.de*

***Abstract*—In modern hardware and system design flows, tight time-to-market constraints can only be met by reusing existing code. Building blocks like floating-point units, embedded processors or bus components are readily available as *Intellectual Property* (IP). However, this practice of putting together third-party components conflicts with the high quality requirements which are common in the domain of safety-critical systems, since the correctness of the used IP blocks is difficult or impossible to verify. In this paper, we propose an approach for safe IP integration by isolating suspicious blocks inside provably safe *container modules*. In this way, system level properties can be checked assuming the correct behavior of the wrapped IP blocks. As a first step in this direction, we show how a container module implementing a bus protocol can be generated and verified automatically. We rely on a model-driven design approach using a domain specific language and model-to-text transformations.**

***Keywords*-hardware design; system level design; safety;**

## I. Introduction

New electronic products are hitting the market every day, ranging from consumer electronics like cell phones to safety-critical embedded systems in transportation. In order to meet time-to-market constraints and to reduce the implementation effort, IP reuse is the key to efficient system design. By putting together available components like processors, arithmetic co-processors, bus components and network interfaces, a lot of the remaining design effort is shifted to software design and system integration. Nevertheless, the quality of the underlying hardware is still a major issue. Even worse, by integrating third-party components, one has to rely on their correctness without being able to verify it.

Especially in the domain of safety-critical systems, verification is not only applied as a post-design quality assurance, but has become an integral part of the whole design process. In order to master the growing complexity of hardware designs, higher levels of abstraction have been introduced, like the *Electronic System Level* (ESL) or the *Formal Specification Level* (FSL [1]). By lifting the validation to these levels, bugs and design flaws can be discovered early, avoiding long cycles of debugging on the lower levels of abstraction. Coverage metrics can be used to assess the quality of the verification. As envisioned in [2], before moving on to the next – more refined – model, complete coverage should be achieved. Such a continuous verification flow establishes a very high level of confidence in the correctness of the created hardware designs.

In order to combine the benefits of IP reuse with the high quality of a verification-driven design flow, we need to find a way to create *correct systems using possibly incorrect components*. This paradigm lifts the creation of reliable systems from unreliable components [3] to the design level. The latter approach takes into consideration electrical defects due to aging and radiation and mitigates possible faults by introducing redundancies on the gate level. However, these techniques assume the functional correctness of the design they are applied on.

On a higher level of abstraction, there are several techniques that have been proposed to enforce *security* properties on the system level [4], [5] by adding supervising components or by adding firewalls in the communication structure of a system on chip. But also these techniques rely on correct components, and no formal proof is given that they really ensure the considered properties. For the problem at hand, they are considering the design at a too abstract level. For instance, it is common that in embedded processors, register outputs are directly wired to address or data outputs to create short critical paths. While this can potentially reveal secret register contents, this is not captured by the mentioned security measures, since the leaking is taking place below the transaction level. For safe IP integration, new approaches need to be developed.

In this paper, we propose to encapsulate potentially harmful IP blocks inside *safe container modules*. The purpose of the containers is to provide a safe and correct interface to the rest of the system. In this way, the overall correctness at the system level can be established in a compositional manner relying on the correct interfaces. A first step is the integration of *monitors* at the interfaces that can passively check the communication of the enclosed IP blocks. However, this requires a supervisor that can react in case an error is signaled and can reset or isolate the suspect. Therefore, on the second level, the container would be able to *filter or alter* outgoing communication in order to correct errors. Finally, on the third and most complex level, the container would

be *actively manipulating* the contained module to force its correct behavior.

In a first implementation, we have created a tool that automatically generates a container component for IP blocks that communicate over a bus protocol. Alongside the RTL code, also verification IP is created that can be used to prove the correctness of the exposed bus interface. By providing a clean implementation of the desired protocol, also register leaking is prevented.

After a brief discussion of related work, we will introduce the general ideas of our approach in Section III. Our first implementation is described in Section IV. We give a short experimental evaluation in Section V before concluding the paper.

## II. Related Work

In [6] and [5], Porquet et al. propose an enhanced memory management in a SoC in order to create secure *compartments* and allow the co-hosting of multiple applications without interference. While the idea of compartments is similar to our containers, their approach requires a specific SoC architecture. Furthermore, like the security measures presented in [4], [7], it assumes the functional correctness of the involved modules.

On a lower level of abstraction, *robustness* or *resilience* is the ability of a system to tolerate faults, either permanent defects or soft errors caused by radiation events [3]. Techniques have been proposed to asses [8] and improve [9], [10] robustness. However, these methods target only a very specific and very low level fault model, while we aim for a more general framework to create correct systems at the design level.

Methods to create circuits which are correct by construction are presented e.g. in [11]–[13]. Starting with a specification in a temporal logic like LTL, an automaton is synthesized that fulfills the spec. However, the involved algorithms have a high complexity. While we do not aim to create full systems from scratch, we plan to investigate the integration of these techniques with our approach, e.g. to automatically generate glue logic between different IP blocks. The construction of *monitors* or *checkers* from assertions is a well-known technique (see e.g. [14]) that can be integrated in our framework.

## III. General Idea

In logistics, the use of intermodal containers (cf. Figure 1) for shipping has been established in the 1950s, and standardized in the 1960s and 1970s. This practice has enabled the swift handling of enormous amounts of goods by standardized procedures worldwide. The ISO container provides a clean interface – no matter what kind of goods are shipped – and allows safe handling and stacking.

These shipping containers provide a good metaphor for the safe integration of IP blocks by creating wrapper modules: Up to a certain level, the wrapper protects the environment from harmful modules inside the container. At the same time, the interface exposed by the wrapper allows the safe composition ("stacking") of modules at the system level.

Figure 1. 40-foot long intermodal shipping container[1]

The goal of the *safe container* approach is to automatically generate wrapper modules for IP blocks that guarantee the correct behavior to the outside world, *independent* of possible design bugs in the contained module. Since the notion of *correct behavior* has a very broad meaning – depending on the nature of the intended behavior – there is no single technique that will cover all aspects simultaneously. In this paper, we have identified three levels of safety, that entail generation and verification techniques of increasing complexity:

1) *Monitoring:* Involves generating hardware checkers that survey the interface of a contained module. In case of a property violation, this requires a trusted master that reacts appropriately.
2) *Filtering & Altering:* Allows to actively ensure certain constraints on the exposed interface, like blocking unauthorized or corrupted communication from a contained module. For incoming signals, a possible mechanism in this category would involve the translation of buggy instructions to equivalent instruction sequences using only trusted functionality.
3) *Active Manipulation:* Involves the intended manipulation of control signals in order to steer the contained module to a correct behavior. This can range from a controlled reset to the triggering of interrupts or the injection of code for active hardware fixes.

## IV. Implementation

As a first proof of concept, we have created a tool for the automatic generation of safe containers for IP blocks that are communicating over an on-chip bus. The tool is implemented on top of the *eclipse* development environment[2], and makes use of *zamiaCAD* [15], an eclipse plugin for the analysis of RTL designs, and *Xtext* [16], a framework for model-driven design that allows the definition of *Domain Specific Languages* (DSLs).

[1] CC BY SA by KMJ via Wikimedia Commons

[2] www.eclipse.org

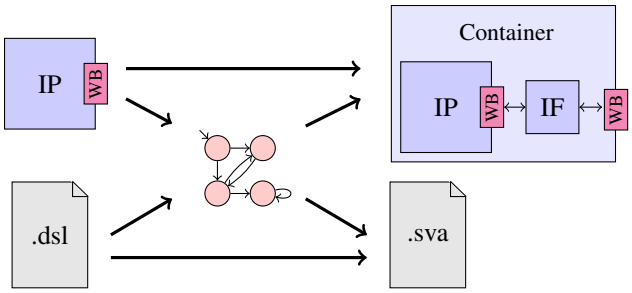


Figure 2. Overall tool flow

### A. Tool Flow

The overall tool flow is illustrated in Figure 2. As input, the target design (in VHDL) and a description of its bus interface (in a dedicated DSL) are given. The interface description is compiled into an extended state machine, which implements the given protocol. After analyzing the target design, the state machine is instantiated with the correct data types and bit widths. Furthermore, an interface module is generated in VHDL. In the next step, the interface FSM is wired up with the target module, forming the final container. Additionally, verification IP is generated in the form of *System Verilog Assertions* (SVA), that can be used to prove the correct functionality of the container interface as well as additional invariants provided in the DSL description.

### B. Domain Specific Language

The bus interface descriptions are given in a dedicated DSL, that has been created using *Xtext*. As an example, a bus master interface for the *Wishbone* [17] protocol is given in Figure 3. In the beginning, all ports belonging to the interface are listed. The actual data types of the respective ports will be determined from the target module during code generation. The central elements in the interface description are the operations. In this simple example, only the two operations `READ` and `WRITE` are specified. An operation is described by means of its *trigger* condition – starting the operation – and its *release* condition, which terminates the operation. Furthermore, for each operation it is specified which ports will be read or written. Following the keyword `stable`, all signals are given that are required to keep their value during the entire operation. This property – and additional invariants specified by the user – will be assembled to assertions (see Section IV-C).

Besides the generic specification of the protocol, design specific constraints can also be added in the interface description. In the example in Figure 3, the expression `su` reflects whether a privileged access to the lowest 1024 addresses is taking place. By adding this constraint to the trigger condition of the write operation, such privileged write accesses will be prohibited.

```
interface wb_master
        ports
                in      CLK
                in      RESET
                fwd in  WB_DAT_I
                fwd in  WB_ACK_I
                fwd out WB_CYC_O = 0
                fwd out WB_STB_O = 0
                fwd out WB_SEL_O = 0
                fwd out WB_WE_O  = 0
                fwd out WB_DAT_O = 0
                fwd out WB_ADR_O = 0
        end

        signal sel    = WB_STB_O & WB_CYC_O
        signal su     = WB_ADR_O < 1024

        operation READ
                trigger   sel & !WB_WE_O
                reads     WB_DAT_I
                writes    WB_ADR_O, WB_SEL_O
                stable    WB_ADR_O
                invariant sel & !WB_WE_O
                release   WB_ACK_I
        end

        operation WRITE
                trigger   !su & sel & WB_WE_O
                writes    WB_DAT_O, WB_ADR_O, WB_SEL_O
                stable    WB_DAT_O, WB_ADR_O
                invariant sel & WB_WE_O
                release   WB_ACK_I
        end
end
```

Figure 3. Wishbone master interface description in DSL

```
READ_WB_ADR_O_stable_a: assert property
  (disable iff (wb_master.RESET)
    (wb_master.STATE == `op_READ) &&
    (!ACK_I)
|=>
    ($stable(ADR_O))
  );
```

Figure 4. Stability assertion for address output during read operation

### C. Verification

Alongside the RTL code of the interface state machine and the container module, also verification IP is generated that can be used to verify the correct behavior of the wrapped module. In particular, for every signal that has been declared as `stable` in the interface description, an assertion is generated. As an example, consider the SVA code in Figure 4. There, it is checked that – unless a reset occurs – during a read operation, the address output will never change before the operation is released by an acknowledge. In a similar way, also user defined invariants are compiled to SVA assertions.

While the stability properties hold by construction of the interface state machine, the assertions act as a formal specification that can be safely relied on when the wrapped module is used in a larger system context.

## V. Evaluation

To evaluate the proposed approach, it has been applied to several hardware designs from *OpenCores*[3], a website hosting open source hardware projects of differing quality and complexity. One of the investigated designs is called *Alwcpu*, "a light weight CPU". This small embedded processor implements a Wishbone master interface. It has been found using a model checker that the CPU contains a bug, which in some cases leads to glitches on the address output. This can lead to register leaking. Furthermore, the effective bus command depends on the delay of the addressed wishbone slave, which is a serious problem and would be hard to track down in a larger system. By enclosing the CPU inside an automatically generated container, the glitchy bus interface is decoupled from the outside world, and the error is corrected. All generated assertions could be proved formally on the container using bounded model checking.

## VI. Conclusion

To enable the design of correct systems from partially verified components, we propose a model-driven approach that encapsulates IP blocks inside safe container modules. These containers are automatically generated from a specification of the correct expected behavior. On three levels of safety, this enables 1) the monitoring of the contained module, 2) filtering and correction of ongoing communication and 3) the active manipulation of the contained module to guarantee functional correctness. In a first implementation, the first two of these objectives have been addressed. In future work, we plan to investigate the possible use of assume-guarantee reasoning [18] and automatic property-based synthesis [11], [12] for more complex scenarios.

## Acknowledgment

This work was supported by the Graduate School SyDe (funded by the German Excellence Initiative within the University of Bremen's institutional strategy) and by the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1.

## References

[1] R. Drechsler, M. Soeken, and R. Wille, "Formal specification level: Towards verification-driven design based on natural language processing," *Forum on Specification and Design Languages (FDL)*, pp. 53–58, Sep. 2012.

[2] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. Le, J. Seiter, M. Soeken, and R. Wille, "Completeness-driven development," in *International Conference on Graph Transformations (ICGT)*, ser. LNCS. Springer, 2012, vol. 7562, pp. 38–50.

[3] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[3] www.opencores.org

[4] J. Sepúlveda, G. Gogniat, R. Pires, W. J. Chau, and M. J. Strum, "Dynamic NoC-based architecture for MPSoC security implementation," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*. ACM, 2011, pp. 197–202.

[5] J. Porquet, A. Greiner, and C. Schwarz, "NoC-MPU: a secure architecture for flexible co-hosting on shared memory MPSoCs," in *Design, Automation & Test in Europe (DATE)*, 2011, pp. 1–4.

[6] J. Porquet, C. Schwarz, and A. Greiner, "Multi-compartment: a new architecture for secure co-hosting on SoC," in *International Symposium on System-on-Chip (SOC)*, 2009, pp. 124–127.

[7] J. Sepúlveda, G. Gogniat, R. Pires, W. Chau, and M. Strum, "Security-enhanced 3D communication structure for dynamic 3D-MPSoCs protection," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, 2013, pp. 1–6.

[8] S. Frehse, G. Fey, E. Arbel, K. Yorav, and R. Drechsler, "Complete and effective robustness checking by means of interpolation," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 82–90.

[9] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Design, Automation & Test in Europe (DATE)*, 2007, pp. 1442–1447.

[10] S. Krishnaswamy, S. Plaza, I. L. Markov, and J. P. Hayes, "Enhancing design robustness with reliability-aware resynthesis and logic simulation," in *International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2007, pp. 149–154.

[11] N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive(1) designs," in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS. Springer, 2006, vol. 3855, pp. 364–380.

[12] R. Ehlers, R. Könighofer, and G. Hofferek, "Symbolically synthesizing small circuits," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 91–100.

[13] K. Morin-Allory, F. N. Javaheri, and D. Borrione, "Fast prototyping from assertions: A pragmatic approach," in *Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2013, pp. 23–32.

[14] K. Morin-Allory, E. Gascard, and D. Borrione, "Synthesis of property monitors for online fault detection," *Journal of Circuits, Systems, and Computers*, vol. 16, no. 6, pp. 943–960, 2007.

[15] A. Tsepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, and V. Tihhomirov, "A scalable model based RTL framework zamiaCAD for static analysis," in *International Conference on Very Large Scale Integration (VLSI-SoC)*, 2012, pp. 171–176.

[16] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. SPLASH '10. ACM, 2010, pp. 307–309.

[17] OpenCores, "WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores (rev. B4)," 2010.

[18] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *International Conference on Computer Aided Verification (CAV)*, 1998, pp. 440–451.