On the Potential of Coq as the Platform of Choice for Hardware Design

Sebastian Ertel, Max Kurze and Michael Raitza
Barkhausen Institute, Dresden, Germany
E-mail: {sebastian.ertel, max.kurze, michael.raitza}@barkhauseninstitut.org

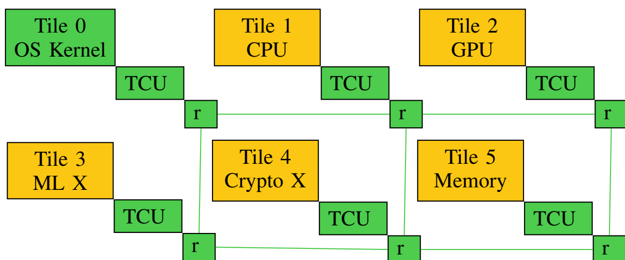
I. Introduction

Hardware is the foundation of our digital world and hence needs to be trustworthy, i.e., correct. Yet most hardware design is done in unsafe languages such as VHDL or Verilog. We believe that Coq provides a promising foundation to put the missing trust into the hardware programs. Several languages with their respective compilers are readily available in the Coq ecosystem [3, 4, 2]. This abstract briefly outlines our experiences so far in bringing Coq-based hardware development to hardware engineers. More specifically, we chose Kôika to re-implement the trusted communication unit (TCU), an essential kernel component in a micro-kernel-based operating system. We report on successes but also obstacles and derive future research directions. The rest of the abstract first presents the TCU design and briefly introduces Kôika. Afterwards, we outline our approach to encourage hardware and operating system engineers alike to write hardware in Kôika. Finally, we report their answers and derive research directions.

II. A Trusted Compute Platform

Hardware/Software co-design customizes systems to performance, energy and security requirements and thereby makes hardware development mainstream. Hardware/software co-design is gaining attention due to several reasons. Increased performance and low energy consumption are among the most promising ones. To meet these requirements, systems became increasingly heterogeneous with custom accelerators (accel) for machine learning, cryptography and many more. The composition of these individual components into a system requires a common interface and new assumptions with respect to mutual trust. The tiled architecture of the M^3 hardware/operating system co-design is one approach to provide both [1, 7]. . We present a typical system composition in M^3 in Figure 1a. M^3 abstracts individual hardware components as *tiles*. A trusted communication unit (TCU) guards the access to each of the tiles and provides a uniform hardware interface throughout the system. TCUs, and respectively tiles, communicate via message-passing over a network-on-chip (NoC) of routers (r).

Semantically, M^3 brings the micro-kernel concept of software-isolated processes to hardware. Each tile represents an isolated hardware component. That is, the effects of faulty or even malicious hardware components are contained to the respective tile. To make sure, that this is indeed the case, the communication in between the tiles is restricted. The kernel (tile) establishes pre-configured communication channels in between the tiles. The TCUs enforce these communication policies during execution. As a result, tiles and their respective hardware components are untrusted. Only the kernel tile, the TCUs and the Noc are trusted. This reduces the trusted computing base (TCB) considerably. Of course, the whole system design only makes sense when the TCB is correct.



(a) Each TCU gates a tile from the router (r) of the Network-on-Chip (NoC). TCUs, the NoC and the kernel tile are trusted. All other tiles are untrusted.

	Timing (ps)		Area			
	WNS	WHS	LUT	FF	DSP	Nets
Verilog	4112	12	3361	1566	1	7541
Kôika	5067	5	3224	1950	0	6603

WNS – worst negative slack, WHS – worst hold slack
LUT – look up table, FF – flip flop, DSP – dig. sig. proc.

(b) Area and timing comparison of Verilog and Kôika designs of the TCU running on FPGA at 100 MHz [10].

Fig. 1: The structure of the TCU-based tiled architecture for trustworthy heterogeneous systems (left) and hardware characteristics in comparison to the original Verilog implementation (right).

```
Example cannot_write_feature_without_kernel_bit:
  let r := init_registers in (* kernel bit is [0] *)
  let inputs := init_inputs in

  run_function r ext_ifaces inputs handle_requests
    (fun ctxt out =>
      get_kernel_bit(ctxt.[FEATURES]) = Ob~0).
  check.
Defined.
```

(a) Test with concrete fixed input.

```
Lemma cannot_write_feature_without_kernel_bit:
  ∀ (reg:reg_t) (r : R reg) (inputs:input),
  get_kernel_bit(r FEATURES) = Ob~0 ->

  run_function r ext_ifaces inputs handle_requests
    (fun ctxt out =>
      get_kernel_bit(ctxt.[FEATURES]) = Ob~0).
  check.
Defined.
```

(b) Lemma reasoning over all possible register states and inputs.

Fig. 2: From tests to lemmas.

III. Kôika in a nutshell

Verification of the whole TCB is a substantial undertaking and hence, we focus on the hardware part first, in particular the TCU. A fundamental property that we seek to very is the isolation of tiles that the TCU enforces. In Coq, several hardware-design-languages with a verified compiler exist such as Fe-Si, Kami and Kôika [3, 4, 2]. We decided to work with Kôika because it is the most recent one. Kôika is a high-level domain-specific language for hardware design embedded in Coq. Kôika comes with a formally-verified compiler (`cuttlec`) that translates Kôika programs into (a subset of) Verilog. The Kôika language is a) *functional* with expressions for variables, binders, abstraction, application and control flow; b) *imperative* where registers store state; and c) *transactional*, to prevent data races. A Kôika program is a set of rules that execute atomically in parallel. That is, racy rules are (deterministically) detected and aborted at runtime.

IV. Lessons Learned and Future Directions

From our experience so far, Kôika seems like a valuable approach to formally-verified hardware design. We have re-implemented a substantial part of the TCU design in Kôika.

Success: Initial evaluation of common metrics for the generated hardware indicate that the re-implementation in Kôika is on par with the original Verilog implementation [6, 5]. Note that the original TCU implementation is written in Verilog, not in BluespecVerilog. Also, the original Verilog-version of the TCU was implemented by hardware experts but co-designed by operating systems engineers (that did not write any hardware). Our Kôika-based TCU re-implementation was developed by electrical engineering and computer science students. That is, language-wise Kôika strikes a very good balance. It replaces reasoning about wires with reasoning about function calls and registers, i.e., state. This reasoning is akin to (imperative) software development. On the other hand, registers are well-known to hardware engineers. Both could quickly grasp the simple and clear structure of Kôika programs. To foster adoption of Kôika, we established a structure derived from unit tests to start from simple but narrow properties to more general ones. In Figure 2, we show an auxiliary property for the isolation theorem of the TCU. On the left (2a), the code initializes explicit inputs and state for registers and only then interprets the function under test. On the right (2b), the same property generalizes over the inputs and fixes only a single register state, the kernel bit. For both cases, the code executes the interpreter on the function `handle_requests` and then verifies an equality for a kernel bit. This property states that a TCU that is not attached to the kernel tile cannot elevate its privileges to become a kernel-attached TCU. Simple reduction in Coq easily discharges both of the goals because Kôika states its semantics as an interpreter, i.e., a reducible function. This easy way to verify hardware code was very well received.

Challenges: But established hardware engineers also raised concerns. In particular, they were worried about Kôika's programming model. The transactional model prevents data races on the one hand but introduces runtime overheads and a novel programming model on the other hand. Ideally, hardware engineers would like continue writing Verilog code that also executes on common HLS tools without Kôika or Coq. The compiled Verilog output of `cuttlec` is naturally nothing that can be manually maintained after all. And indeed, a better integration of the Kôika EDSL into Coq is very much desirable. Even though the **`Notation`**-based integration worked well, we faced problems when porting Kôika to the latest version of Coq due to changes with respect to **`Notation`** [8].

Future Directions: We believe that the Coq ecosystem already provides interesting projects to overcome these concerns. Instead of **`Notation`**s, MetaCoq may provide a much better framework to implement future DSLs [9]. For Kôika, it would provide for much better experience when defining registers or data types like `structs`. Indeed, Kôika could be aligned much closer to Verilog. Taking it to the extreme: after all both Verilog and Coq files have the same `.v` as their file suffix[1]. But in order to align the Kôika programming model closer to the one of Verilog, the transactional model needs to be replaced. Transactions are a relict of Bluespec to make unsafe Verilog programs data race free. In a powerful environment such as Coq, it is certainly possible to verify that my Kôika program is free of data races. These are only some of the directions that we intend to pursue in the future.

[1]This sentence is of course to be taken with a grain of salt.

References

[1] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. “M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores”. In: *SIGARCH Comput. Archit. News* 44.2 (Mar. 2016), pp. 189–203. ISSN: 0163-5964. DOI: 10.1145/2980024.2872371. URL: https://doi.org/10.1145/2980024.2872371.

[2] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. “The essence of Bluespec: a core language for rule-based hardware design”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 243–257. ISBN: 9781450376136. DOI: 10.1145/3385412.3385965. URL: https://doi.org/10.1145/3385412.3385965.

[3] Thomas Braibant and Adam Chlipala. “Formal Verification of Hardware Synthesis”. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 213–228. ISBN: 978-3-642-39799-8.

[4] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. “Kami: a platform for high-level parametric hardware specification and its modular verification”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110268. URL: https://doi.org/10.1145/3110268.

[5] Sebastian Haas and Nils Asmussen. “A Trusted Communication Unit for Secure Tiled Hardware Architectures”. In: *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2022, pp. 1–4. DOI: 10.1109/ICECS202256217.2022.9971056.

[6] Friedrich Pauls, Sebastian Haas, and Mattis Hasler. “Trust-minimized Integration of Third-Party Intellectual Property Cores”. In: *2023 20th International SoC Design Conference (ISOCC)*. 2023, pp. 53–54. DOI: 10.1109/ISOCC59558.2023.10396198.

[7] Friedrich Pauls, Sebastian Haas, Stefan Köpsell, Michael Roitzsch, Nils Asmussen, and Gerhard Fettweis. “On trustworthy scalable hardware/software platform design”. In: *Smart Systems Integration* (2022).

[8] Clément Pit-Claudel and Thomas Bourgeat. “An experience report on writing usable DSLs in Coq”. In: *7th International Workshop on Coq for Programming Languages (CoqPL 2021)*. 2021.

[9] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. “The metacoq project”. In: *Journal of automated reasoning* 64.5 (2020), pp. 947–999.

[10] Oguzhan Türk. “A formally verified Hardware Design of a Communication Unit in a Micro-Kernel Operating System”. MA thesis. University of Technology, Dresden, Germany, 2022. URL: https://github.com/Barkhausen-Institut/tcu-koika/blob/main/documentation/Report/Thesis/Turk_Oguzhan_Master_Thesis.pdf.