CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Thesis for the Degree of Licentiate of Engineering

Building Verified Hardware and Verified Stacks in HOL

Andreas Lööw

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Gothenburg, Sweden
2019

Building Verified Hardware and Verified Stacks in HOL

Technical Report 194L
ISSN 1652-876X
Department of Computer Science and Engineering

Chalmers University of Technology and
University of Gothenburg
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2019

Abstract

This thesis explores building provably correct software and hardware inside the HOL4 interactive theorem prover. Interactive theorem provers such as HOL4 are proof environments where manual (human) and automated (machine) proofs can be composed in logically safe ways, and all proof steps (be it manual or automated) are mechanically checked.

We are in particular interested in systems consisting of both software and hardware. Such systems are of relevance when building so called verified stacks. A verified stack is a computer system accompanied by a correctness theorem ensuring the correctness of its running software down to the computer system's hardware implementation.

Our foremost contribution to hardware development in HOL4 is a proof-producing tool for extracting hardware circuits proved correct inside HOL4 to the hardware description language Verilog. Verilog is one of the low-level description languages modern synthesis tools take as input. By targeting Verilog, we can close the gap between the mathematical models of proved-correct circuits inside our prover and the input to the synthesis tools that will ultimately be used in the creation of concrete hardware, e.g., FPGA artifacts, out of our hardware descriptions.

Our contribution to the tradition of building verified stacks is as follows. We have used our Verilog translator tool in the construction of a verified proof-of-concept processor that we have extracted and synthesized for an FPGA board. Building upon this work, we have used the processor as the hardware basis for verified stacks based on CakeML programs, including a stack for compiling CakeML programs and a stack for checking proofs. To be able to construct such stacks, we adapted and extended the verified CakeML compiler and its development methodology to support targeting the new processor we have constructed. The CakeML compiler previously only supported compilation to x86, ARM and other architectures without a verified implementation.

Contents

Chapter 1

Introduction

This thesis concerns building trustworthy computer systems. Two goals have guided our work:

Goal 1 For (computer system) correctness arguments, we are to rely only on *proof that hold up to the highest of standards*: Specifically, formal proof mechanically checked by a theorem prover called HOL4.

We discuss the HOL4 theorem prover in more detail in Sec. 1.1.

Goal 2 We should be satisfied with nothing less than *fully verified computer systems*: That is, computer systems where each individual component has been shown to be correct, by aforementioned HOL4 proofs, and the components and their correctness proofs, together, form a correct computer system (a "verified computer system").

More details on verification are given in Sec. 1.2, and more details on verified stacks (verified computer systems) are given in Sec. 1.3.

The two goals **Goal 1** and **Goal 2** are ambitious goals. But the worried reader need not worry: We have been able to reach both goals. In the two papers included in this thesis, included as Chap. 2 and Chap. 3, HOL4 proofs are used to demonstrate the correctness of both hardware and software, including systems consisting of both. In particular, in Chap. 2, we show how we have built fully verified computer systems, capable of tasks such as compiling programs and checking proofs. Of course, this is not to say that Intel, AMD, and all other processor and computer developers will now go out of business: The systems we have built are research prototypes – but nonetheless, the systems satisfy both of the goals above, and can be used as groundwork for future, larger, verification projects. The two papers are outlined in more detail in Sec. 1.4. Lastly, this introduction chapter ends with Sec. 1.5, where we evaluate our achievements based on a set of criteria part of a verified stack challenge set out by Moore [46].

In the rest of this section, we contrast the contents of this thesis, and the two goals **Goal 1** and **Goal 2**, against modern-day real-world software and

hardware development. To be more specific, it is no difficult task to come up with examples of real-world software and hardware bugs, consider e.g. the bugs enumerated by Simson [18], or, more recent bugs such as Heartbleed [1], and Meltdown and Spectre [2]: How does the work in this thesis relate to such bugs?

Today, most real-world systems are developed under the paradigm of testing-based assurance, rather than proof-based assurance, as done in this thesis. Testing is the practice of stimulating a system you have built with input (possibly well thought out input, possibly random input) and then checking if the system reacts appropriately.

One major problem with testing-based assurance is that most real-world systems have a too large input space for it to be feasible for any testing-based assurance approach to exhaustively explore the entire space. Assurance approaches based on mathematical proof, which we employ in this thesis, in contrast, suffer from no such limitation, and can – if needed – cover even infinite input spaces. This is an important advantage of proof-based approaches, as if we do not cover the entire input space of a system, we run the risk of not detecting bugs lurking in untested parts of the system.

Another advantage of proof-based assurance compared to testing-based assurance is that in proof-based assurance the system to be verified (often) has to be described in multiple formalisms (often, one formalism for the "specification" of the system and one for the "implementation" of the system, see Sec. 1.2), and this forces us to see the system from multiple perspectives, and this in itself sometimes lead to the discovery of design faults or other "suspicious" aspects of a system.

Based on these advantages, our claim is, then, if we mathematically proved the absence of bugs in our real-world systems, i.e. employed proof-based assurance, rather than solely relied on testing-based assurance, then more real-world bugs would be caught in development rather than in deployment. Relating back to the bugs referred to above, in the same way as there are different kinds of tests, such as functional tests and performance tests, there are different kinds of proofs. In this thesis we focus exclusively on proofs of functional properties of systems. Given this, we should note that some of the well-known bugs referred to above are not functional bugs, but rather e.g. security bugs. Security bugs are not directly addressed by this thesis. This is a limitation in scope, not a limitation of proof-based assurance. The work here could function as a basis for security work, but we do not speculate on such extensions. To be more precise, our claim concerns functional properties: By the advantages listed above, proofs of functional properties have the potential to discover more functional bugs than any collection of functional tests can.

Goal 1 relates to what kind of proofs we accept as proofs: We set the bar high, and only accept HOL4 proofs. **Goal 2** relates to the scope of system verification we are interested in: Here we also set the bar high, as we do not simply want to verify a "critical part", or otherwise especially suspicious parts, of a particular system, but rather entire computer systems – this includes all layers, from software down to hardware.

1.1 Interactive theorem proving

To meet **Goal 1**, introduced in the very beginning of this chapter, to rely only on proofs satisfying the highest of standards, all developments for this thesis have been carried out in the HOL4 interactive theorem prover [44]. This section provides a short description of HOL4, and compares its interactive theorem proving approach to other theorem proving approaches.

HOL4 is an implementation of higher-order logic (HOL), and is built in the LCF tradition [20]. For LCF-style provers, to trust the validity of the theorems proved with help of the prover (or in other words, trust our proofs), we need only to trust a small kernel in the center of the prover. The kernel contains primitive inference rules and other foundational logical machinery. That we only need to trust the kernel is a slight oversimplification, but is largely true. A more detailed discussion on what is needed to trust an LCF-style prover would have at least also included trust issues related to parsing, pretty-printing and the prover runtime environment. Pollack [43] discusses these, and other, trust concerns in more detail. We will address concerns similar to these concerns, at least indirectly, in the upcoming sections (in particular runtime concerns).

In HOL4 (and other LCF-style provers), all proof infrastructure builds upon the prover's kernel, in a way that extends the capabilities of the prover without extending the trusted based of the prover. Both tactic-based interactive and automated proof techniques have been built upon this kernel, and the LCF-style architecture allows techniques of both types to be utilized in the same proof in logically sound ways. This enables a cooperative-based conversation style between the user of the prover and prover (between human and machine) where the user is responsible for big-picture proof management and the prover takes care of routine intermediate proof steps. Furthermore, proof work carried out by humans, automated proof-search procedures and other unreliable beings or programs, are always checked by the kernel. That is, to trust a HOL4 theorem, you only need to trust the HOL4 prover, not the human that used the prover to prove the theorem in question.

Interactive theorem proving combines the strengths of human and machine reasoning. Other theorem proving approaches, such as pen-and-paper proving

and so-called automated theorem proving, do not provide composition stories for human and machine reasoning.

In pen-and-paper mathematics, machines are sometimes used for aid (e.g., for uninteresting computations or for explanatory aims), but pen-and-paper mathematics do not have a composition story for human and machine reasoning: The result of human reasoning is stored as proofs in formats not readable by machines, so, consequently, human reasoning cannot be checked by a machine. And machine proofs cannot be easily checked by humans, as machine proofs are rarely comprehensible through direct consumption by humans.

In automated theorem proving, we face similar problems, as we also there are missing a convincing human and machine reasoning composition story. Most real-world problems are too big and complicated for direct automated theorem prover consumption, and must therefore be simplified, transformed, and decomposed ("prechewed") outside the formal systems offered by automated provers before any such prover can handle them. The operations carried out outside a prover's formal system, be it by humans or other programs, cannot be checked by the prover.

Continuing our comparison between interactive and automated theorem provers, another thing to note about automated theorem provers is that they are rarely as automatic as they are sometimes made out to be: Rather, an intimate alliance between human and machine needs to be formed for the automated prover to reach its full potential – but conversations between humans and automated provers lack the expressiveness and flexibility found in conversations between humans and interactive provers. When an automated prover needs user guidance to prove a difficult goal, the user is often limited to a crude and awkward conversation with the automated prover where the only way to offer human intuition to aid the prover's proof search is through hints expressed in terms of obscure configuration flags. If attempts based on tweaking configuration flags are unsuccessful, the user has to reformulate the problem or split the problem into smaller pieces (i.e., carrying out more "prechewing" outside the prover's formal system).

More detailed background on interactive theorem proving and LCF-style provers have been given elsewhere [19, 24, 20, 21].

1.2 Verified software and hardware

In this section we make more precise what we mean by something being "verified". In other words, we clarify **Goal 2**, introduced in the beginning of the chapter, saying that we are interested in "verified" computer systems, by elaborating on what we mean by "verified".

That a piece of software or hardware F is "verified" can mean widely different things. In this thesis, that F is verified will always mean that it is verified *with respect to something*. In more precise language, verified will be taken to mean that F behaves according to an idea of how F ought to behave. Here, let us call this idea S. We will sometimes refer to S as a specification for F. With this terminological setup, that "F is verified with respect to S" means, simply, that "F satisfies S" (for some (precise) meaning of satisfies).[1]

Of particular interest to us is what it means for something to be verified inside HOL4, as, as already stated, all developments in this thesis have been carried out in HOL4. When working with HOL4, we are limited to reason about mathematical objects. But neither F nor S are mathematical objects! For example, if F is a piece of hardware, it is a physical object. And S might be just an idea inside our head. Consequently, we need a mathematical representation of both F and S. Here, let us call these f and s, respectively.

We now have four different entities to keep track of: F, f, S, and s. They are related in the following ways: (1) f is supposed to model F, (2) s is supposed to model S, and (3) f is supposed to satisfy s. Out of these three, only (3) is a relation between two mathematical objects. This means that only (3) can be proven inside HOL4, whereas the others, (1) and (2), are informal relations, in the sense that it is impossible to prove that they hold inside HOL4 (or inside any other theorem prover, because in theorem provers, we can only reason about mathematical objects).

We can, with some confidence, say that F satisfies S (our target claim to show in a verification project) if we can provide convincing evidence for all three above relations. The kind of evidence we will provide for relation (3), the relation between f and s, is unproblematic: The evidence will be in the form of HOL proofs, checked by our trustworthy HOL kernel. For the relations (1) and (2), the answers are more involved.

For relation (1), one important question to ask is at what level f models F. In other words, at what abstraction level does f live? For example, if F is a piece of software, two possible levels are the source code level and the machine code level. Verifying software at the source code level is simpler, but does not model the actual code that will be run by users of your software: users will run the machine code representation of your code. Similarly, for hardware, different abstraction level targets are possible. Do we target a (hardware-level) behavioral description of our system, or is our system represented as a set of connected gates (like and, or, not gates, etc.) and flip-flops?

For relation (2), the questions to be asked concern s and S instead of f

[1] The exact notation introduced here, with F and S, is only for explanatory purposes, and is not used in the included papers; but the underlying ideas in the papers are the same as the ideas explained here. The same hold for other notation introduced in this chapter.

and F. For example, at what level of detail does s model the behavior we are interested in (that is, S)? Or, does s capture all behavior we are interested in, or just a "critical part" of what we are interested in? To exemplify, if we are specifying a piece of hardware's behavior, are we interested in the exact timing behavior of the hardware, or is it sufficient to demand e.g. that a result should eventually be computed, without specifying the exact number of clock cycles the computation should take?

What the "right" answers to these and other questions about (1) and (2) are depend on context. The core problem is that the more details that are modeled in f and s, the more effort is needed to show that f satisfies s ("verify f"). At the same time, the more details that are modeled, the more valuable the verification result, as the result will cover a larger part of F and S and the correspondence between s and S and f and F are clearer if more details are modeled.

In summary, the more detail in our mathematical models, the more valuable the verification result, but also the more expensive the verification result. Clearly, there is a trade-off here. To be able to give an answer to how much detail is to be included, the context of our verification work must be taken into account. How much time do we have? How much resources (money, expertise, etc.) do we have? How important is it that what we build is actually correct? For example, most people will agree that it is more important that autonomous cars function correctly than e.g. that mobile phone games function correctly.

1.3 Verified stacks

In this section, to further clarify **Goal 2**, we discuss some verification aspects of building systems from smaller pieces. In the previous section we considered one piece of software or hardware F. We said that to be able to carry out verification, F needs to be modeled as a mathematical object f inside our theorem prover. Put this way, it sound as if f should be one single monolithic thing. When F is something small, like a simple adder circuit, this might make sense. But when F is something more complex, like a complete computer system, treating f as a monolithic thing becomes impractical.

When F is a complete computer system, it makes sense to think of F as something that can be decomposed into smaller pieces that together form F. Often, the pieces we decompose F into can themselves be thought of as pieces that can be decomposed into even smaller pieces. One example decomposition of a computer system is into a software part and a hardware part. Obviously, these two parts can themselves be decomposed into smaller parts. For example, the software part may consist of application software, an operating system, etc.

A good decomposition of f into multiple pieces, say $f_0, f_1, \ldots, f_n$, opens

up the possibility for the development and verification of each piece to be carried out fairly independently of other pieces, possibly even by different development teams. But the pieces can never become fully independent: In every step of development we must keep in mind that the pieces, ultimately, must fit together. They must fit together both in a functionality sense, and in a verification sense. That is, together the pieces must offer the entire functionality of f ("functionality sense") and the verification result for each piece must together form a verification result for f ("verification sense").

Remember that we have a specification s for f. To be able to independently verify each piece of f, we need a specification s_i for each $f_i, 0 \leq i \leq n$. For example, consider a very simple computer system capable of only one thing: saying "hello world!" (this is our s). This simple system can be decomposed into two parts: one software part f_0, the hello world program, and one hardware part f_1, the model of the physical computer (i.e., f_1 is the implementation of the computer expressed as some abstraction level, e.g. at the gate level). In this decomposition, s_0 will say that f_0 prints "hello world!", and s_1 will say that f_1 implements the computer's instruction set architecture (ISA). In this scenario, to be able to compose our two layers, f_0 must be implemented in terms of the ISA of the computer. That is, f_0 must be verified down to machine code. With f_0 at this level, $f_0, s_0, f_1,$ and s_1 can be composed into the theorem we want: f satisfies s.

Notice that, in a very concrete sense, f_0 is stacked on top of f_1 in the above example to form f (put another way, we run f_0 on top of f_1, by loading f_0 into f_1's memory). The same kind of composition by stacking occurs when f is split into more than two pieces. This explains why we sometimes refer to systems verified as in the example above as a verified stack.

These kinds of decompositions are not only a theoretical possibility, but can also be carried out in practice. This is, in fact, what we have done in the first paper included in this thesis. There we have decomposed computer systems into multiple layers, verified each layer, and have composed them together to form coherent pieces, such as a verified stack for compiling programs. The contents of this paper, and the other paper included, is outlined in more detail in Sec. 1.4.

The stacks we have built are not the first verified stacks to exist: Two earlier projects with aims similar to ours are the CLI stack project [46, 6] and the Verisoft stack project [4]. Both projects, now ended, have constructed verified stacks, including verified processors and verified compilers for their processors (but with limitations, which we address in the first paper included in this thesis). There is also the ongoing DeepSpec project [5]; the effort is "anticipating a world where large verified systems are routinely built out of smaller verified

components that are also used by many other projects." They emphasize, what we also think should be the aim of stack projects, that their aim is not to build one specific verified stack, but rather to construct or enable an "engineering discipline where novel artefacts can be predictably developed based on existing components, standardized methodologies and a well-educated workforce." But at the same time, as they also point out, concrete and functional verified stacks are needed as evidence that the methods they (and we) put forward in their (and our) papers work not just in theory but also in practice.

One distinct advantage over other approaches, such as traditional informal system building, of building verified stacks inside interactive theorem provers is that the trusted bases of the stacks are kept to a minimum. For any specific stack, we must still provide evidence outside the formal system our theorem prover offers to support the connection between s and S (that the specification is "correct") and f and F (that the model is "correct"), but the connection between s and f does not extend the trusted base of the final stack. For example, in the hello world computer example above, the ISA used to glue f_0 and f_1 together does not extend the trusted base of the example stack, as the ISA is only part of the proof of the connection between s and f, and not s or f themselves, and the proof does not affect the trusted base. Generalizing from this, the intermediate layers of the stack do not affect the trusted base: only the, in some sense, "top part" (the specification) and the "bottom part" (the model of hardware (e.g. model of gates) used) of the stack end up in the trusted base.

Beyond the intermediate layers disappearing from the trusted base, by building stacks, we also learn that the intermediate specifications are both implementable and useful. These are two important properties of any specification. For example, the ISA from the computer example above is an intermediate specification in the example. If we would have stopped our verification of the hello world program at the ISA level, then we would have only showed that the ISA is useful. But by carrying out the verification of the hello world program down to the hardware implementation of the computer, we have as an intermediate step showed that the ISA is also implementable. This means that the ISA, as a specification layer in the stack, has passed two challenges: First, the ISA is comprehensive enough that it is possible to verify the hello world program against it, and, secondly, the ISA is realistic enough that it is implementable. Both being useful and comprehensive, and being realistic and implementable are important properties of specifications, and including a specification as an intermediate layer in a stack puts it up for the challenge of being both simultaneously. But we must also stay clear of exaggerating what we learn from including a specification as an intermediate layer in a stack; we learn that the specification is "useful" in one particular situation, and from

this it does not follow that the specification is useful in general. For example, it is possible that only a small fraction of the ISA was needed to verify the hello world program in the above example; or that the ISA is overfitted for the particular situation it was used in. But, still, we learn that the specification in question is useful in at least one context, and if this context is complex or we have some other reason to believe that the context is representative of most or many other contexts the specification will be used it, then, at least, we have some concrete backing for the claim that the specification is "useful" in general.

1.4 Contribution of included papers

This thesis includes two papers describing how one can go about developing provably correct hardware and software inside the HOL4 interactive theorem prover, and this section highlights some of the contributions of the papers. The papers included in this thesis are identical to the versions of the papers to be published in the proceedings of the respective conferences they have been accepted to.

The two papers considered together make the following contributions:

- First and second paper: We present a hardware development methodology that is centered around functional versions of Verilog programs in HOL. Verilog is a well-known low-level hardware description language, and is useful for e.g. synthesizing for FPGAs.
- First and second paper: We present an automatic translator tool that makes the hardware development methodology have a solid connection to an explicitly defined operational semantics for Verilog we developed in parallel with the tool.
- Second paper: The description of our formal semantics for a subset of Verilog is also a contribution. The semantics is carefully carved out to be faithful to the Verilog standard, manageable in complexity for HOL proofs, and sufficiently large to express interesting synthesizable hardware.
- First paper: We show how we have used our hardware development methodology to construct and verify a simple processor down to its implementation in Verilog.
- First paper: We exhibit sufficient properties that, if proved about a compiler and a processor, enable them to be used together for constructing verified stacks. As part of this, in concrete terms, we have extended the trustworthy development methodology of the CakeML project, including its verified compiler, with a connection to our new processor.

- First paper: We construct stacks verified down to Verilog for, amongst others, compiling (CakeML) programs and checking proofs. We have, furthermore, run these stacks on an FPGA board.

- First paper: By including the CakeML compiler in stack constructions, we provide evidence for earlier claims made in the CakeML project that the assumptions on the CakeML compiler's correctness theorem are reasonable (satisfiable/"realistic").

1.4.1 First paper: Verified Compilation on a Verified Processor

The first paper, "Verified Compilation on a Verified Processor", puts forward a methodology for building verified stacks. As described in Sec. 1.3, a verified stack is a computer system accompanied by a proof demonstrating its correctness. A computer system consists of both hardware and software. In stacks produced by our methodology, the hardware part is fixed and consists of a verified processor developed with the help of a translator tool – presented in more detail in the second paper included in this thesis – enabling hardware development inside HOL4. We have synthesized the processor for an FPGA. The software part of the produced stacks, on the other hand, is to be varied freely: Any CakeML program can be compiled to the verified processor, just like it could have been be compiled for e.g. (unverified) x86 and ARM processors. If we have a verified CakeML program, i.e. a program that has been proven to satisfy a specification, then we can compose the correctness theorems of the processor and the CakeML program in such a way that we get a theorem stating that the compiled CakeML program when run on the processor satisfies the same specification as the original CakeML program. Or, in other words, we can build a verified stack offering the functionality of the CakeML program as an independent computer system.

Statement of contribution. For the paper, I designed, developed and verified the processor. I was also responsible for the lab hardware setup, including FPGA development and surrounding tooling. I was involved, from the hardware side, in the integration of the CakeML compiler and the processor (e.g., adapting the processor and its surrounding components for the integration), but the pure software work was carried out by the other authors of the paper.

1.4.2 Second paper: A Proof-Producing Translator for Verilog Development in HOL

The second paper, "A Proof-Producing Translator for Verilog Development in HOL", provides further details on the tool and its associated hardware development methodology used in the first paper to build the processor. The tool is proof-producing and translates hardware descriptions from HOL to the hardware description language Verilog. That is, given a hardware description in HOL – which can be used in proofs in HOL – the tool gives back both a corresponding hardware description in Verilog – which can be used as input to hardware synthesis tools – and a HOL theorem ensuring that the input (i.e., the HOL description) and output (i.e., the Verilog description) describe the same circuit (i.e., both exhibit the same behavior). Beyond ensuring that the translation is correct, the generated HOL theorems can also be used to transport correctness results about input HOL hardware descriptions to their respective output Verilog hardware descriptions.

Statement of contribution. For the paper, I alone have developed everything described by the paper. My co-author gave advice and adjusted the presentation in the paper.

1.5 Moore's verified stack challenge

In his paper "A Grand Challenge Proposal for Formal Methods: A Verified Stack"[2] [46], Moore sets out a challenge to build a verified stack. The challenge, as defined in the abstract of his paper, is to "build and mechanically verify a practical computing system, from transistors to software." In his paper, Moore makes the challenge more precise by breaking it down into a list of criteria to be met (which we return to shortly).

For this section, we evaluate what is achieved (with respect to artifacts) in the first paper included in this thesis with respect to the challenge defined by Moore. (During our work, however, it has not been a goal to take on the challenge. That is, the challenge has not guided our work; nonetheless, we found it instructive to, after our work was complete, evaluate our work against Moore's challenge.) In what follows, we provide a short evaluation with respect to each of the challenge's criteria.

The project should produce an artifact, e.g., a chip. This criterion is, in at least one sense, met. We have synthesized our processor for an FPGA board, and, furthermore, we have also run non-trivial programs such as the

[2]Formal methods is the discipline this thesis belongs to.

CakeML compiler on top of the FPGA instance of the processor. Synthesizing the processor was possible as our Verilog translator tool produces synthesizable Verilog, meaning that Verilog synthesis toolchains can "compile" the output Verilog code to an FPGA bitstream that can be used to configure FPGA chips. For the synthesis of the processor, we used the Vivado Design Suite from Xilinx.

On the other hand, while we have produced a physical artifact, we must be precise with what we mean by artifact here, and how this artifact relates to our formal proofs. As the challenge is defined, a stack successfully passing the challenge must be verified "from transistors to software". However, our proofs do not reach all the way down to transistors. Rather, our proofs reach down to a formal semantics of process-based Verilog that we have developed in parallel with the Verilog translator. To be able to load the Verilog implementation of our processor onto our FPGA board, the Verilog implementation must first be transformed to an FPGA bitstream by an unverified tool (in our case): the Vivado Design Suite. One limitation, thus, of our study is that we do not target the transistor level.

At the same time, it is worth pointing out that even when targeting abstraction levels below process-based Verilog, questions regarding what the verification result actually ensures will remain (in the terminology of Sec. 1.3, the relation between F and f is always informal). For example, even when targeting a gate-level abstraction level, one could still raise concerns. For example, Moore [45], in an earlier paper, formulates one such concern as: "Of course, two almost philosophical questions remain: Does the formal model at the gate level accurately reflect reality? [...] These questions are nontrivial." Klein [30] formulates a similar concern as: "Describing [physical artifacts] precisely is the realm of physics, and if pursued to the end, the question of whether we can even describe and predict reality in full detail is a matter of philosophy." Nevertheless, raising these concerns is not to say that a verification result with a stack capable of running non-trivial programs (with proofs all the way), targeting the gate level, or even the transistor level, would not be a stronger result than what is achieved in our paper.

The artifact's behavior should be of interest to people not in formal methods. This criterion is not met. We can run useful programs on top of our synthesized processor, such as the CakeML compiler and a proof checker; however, the processor is too slow for industrial usage, i.e. too slow for people not in formal methods to have any practical use for the stacks produced by our methodology.

The behaviors of the artifacts/stacks we have produced are currently only of interest to people not in formal methods to the extent the behaviors illustrate what can be achieved with today's formal methods. But it should be possible to

subject our results to modular improvements, in a fashion similar to how the CakeML compiler has been modularly improved during its lifetime. As long as layer boundaries remain the same, stack layers can be improved independently of each other. For example, if one would verify a pipelined version of the processor and show that it satisfies the same ISA as the current non-pipelined processor, then one could build stacks based on the pipelined processor instead of the current processor without having to revisit the verification work already carried out at and in between the non-processor-implementation layers, such as the software-hardware integrations (e.g., the CakeML backend for the processor).

Of course, one cannot modify, or otherwise vary, a verified stack in such a way that it is made more relevant to practice without, simultaneously, also affecting other aspects of the stack. Making a stack more "practical", in most cases, means also making it larger or in some other way more complicated (e.g., by re-implementing existing components in a more optimized way). This means that the verification of the stack, which relates to e.g. the next criterion evaluated, will become more involved as a consequence. In this sense, the criteria are interrelated. (This observation has consequences such as the insight that counting the number of met criteria is not a good way to summarize how close to meeting the challenge as a whole a specific stack is.)

The artifact should come with a "warranty" expressed as a mathematical theorem. This criterion is met. For any CakeML program, using our methodology a warranty theorem can be produced stating that the CakeML program in compiled form running on our processor (at the Verilog level) will exhibit the same behavior as the original CakeML program. Furthermore, if the CakeML program in question is verified, the guarantees proved about the program can be transported to the compiled CakeML program running on top of the processor. In such cases, we can talk about the composition of the compiled CakeML program and the processor as a verified stack.

The warranty should be certified mechanically; user input and interaction are allowed but mechanical checkers must be responsible for the soundness of the claim. This criterion is met. HOL4 is used to mechanically certify all developments presented in this thesis. Furthermore, as HOL4 builds upon an LCF-style architecture, HOL4 is a high-assurance proof checker.

The criterion is phrased in terms of checkers, rather than a single checker. Moore expresses some concerns related to relying on a single tool, thereby putting all of one's eggs in one basket, rather than taking advantage of the diverse set of special-purpose tools developed by different scientific communities.

We rely, as stated, on only a single checker: HOL4. But HOL4 is a versatile basket. In fact, the restriction of using HOL4 as one's proof checker, does not

restrict one's array of techniques that can be employed for finding proofs: As long as the tool or technique in question on can produce HOL proofs, any tool or technique can be used. For example, in our correctness proof for our processor, a SAT solver was used to find HOL proofs for some low-level but complicated identities involving operations over bit vectors. In other words, a restriction to HOL4 does not imply any restrictions on which techniques can be applied in verification work. Actually, it is a necessity to not rely on more than one proof checker if we want to keep the trusted base of the stacks we build as small as possible.

The checkers used to certify the warranty should be available for others, at least, to use and test, if not inspect. This criterion is met. As stated in the evaluation of the previous criterion, the only checker used is HOL4. HOL4 is open source software, meaning that all of its source code is publicly available. Furthermore, the source code for the developments from the papers in this thesis are publicly available (including the extended CakeML compiler and the Verilog translator).