Is "Implementation Implies Specification" Enough?

Paul E. Black
National Institute of Standards and Technology
100 Bureau Drive, Stop 8970
Gaithersburg, Maryland 20899-8970 USA
paul.black@nist.gov

September 21, 1999

Abstract

An implementation is typically checked against a specification by proving that the implementation implies the specification. This ensures that the implementation only has behaviors allowed by the specification. However, this does not require the implementation to have any behavior at all!

We propose that correctness statements have two parts, corresponding to *liveness* and *safety*. Safety is that the implementation implies an "allowed-behavior" specification, as now. Liveness is that a "required-behavior" specification implies the implementation.

1 Introduction

The current practise to formally check an implementation against a specification is to set up and prove a correctness statement that the implementation implies the specification:

$$D_{impl} \Longrightarrow D_{spec}$$

Informally, this states that any behavior of the implementation is allowed by the specification, see for instance [1, 2, 4]. Implication is used since equality between the implementation and the specification is too confining. Equality may require the specification to have too many implementation details.

For example, consider an address decoder for a register file as in Fig. 1. One of three addresses, 00, 01, or 10, comes from above on the address lines a0 and a1. The decoder selects one of three registers with r0, r1, and r2. The selected register puts data onto or receives data from the data lines at the bottom. (The register read/write controls and clock lines are not shown.) We want the decoder to consistently activate *some* register select line in response to each address, but it doesn't matter to correct operation which one is selected. Since we want to allow the layout designer the maximum flexibility, we will not specify the mapping between addresses and registers.

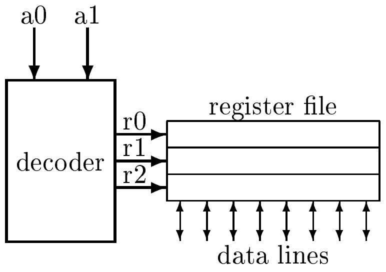

Figure 1: Address Decoder for a Register File

Here is a specification written in HOL98 [3]. The function being defined, `decode_spec`, and its arguments are given first. Question mark (?) is "there exists" ($\exists$), period (.) is "such that," leaning slashes (/\) is "and," and `EL n` selects the n^{th} element of a list (`[...]`).

```
decode_spec a0 a1 r0 r1 r2 =
 ? ra rb rc . ~(ra = rb) /\ ~(ra = rc) /\ ~(rb = rc) /\
  (~a0 /\ ~a1 = EL ra [r0; r1; r2]) /\
  (~a0 /\  a1 = EL rb [r0; r1; r2]) /\
  ( a0 /\ ~a1 = EL rc [r0; r1; r2])
```

Informally this says that each address is tied to one of three outputs. The correspondence of addresses to outputs, determined by ra, rb, and rc, is not fixed, but each address must correspond to a unique output.

Since one possible address is not allowed, we also define a constraint on valid inputs to the decoder.

```
decode_constraint a0 a1 = ~(a0 /\ a1)
```

The traditional correctness statement is as follows.

```
decode_constraint a0 a1 /\ decode_impl a0 a1 r0 r1 r2 ==>
    decode_spec a0 a1 r0 r1 r2
```

The problem is that this correctness statement doesn't require the implementation to do anything. If the implementation is contradictory, that is, it evaluates to false, the implication is true no matter what the specification requires. In other words, a contradictory or inconsistent implementation model satisfies any specification!

2 Proposed Correctness Statement

Can this problem occur in practise? Figure 2 shows one implementation of the decoder. The formal description is simply a list of gates and connections. (Appropriate definitions of not_gate and and_gate are needed, but are not given here.)

```
decode_impl a0 a1 r0 r1 r2 =
  (? ca cb cc cd .
    not_gate a0 ca /\ not_gate a1 cc /\
    not_gate ca cb /\ not_gate cc cd /\
    and_gate ca cc r0 /\
    and_gate ca cd r1 /\
    and_gate cb cc r2)
```

In this slightly contrived example, the address inputs, a0 and a1, are initially buffered resulting in negated signals, ca and cc. The negated signals are inverted again to get true signals, cb and cd (cd not labeled in figure).

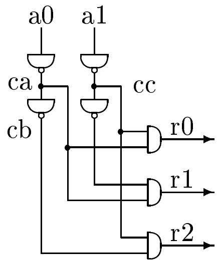

Figure 2: An Implementation of the Address Decoder

Suppose an error were made in the formal model and the second line was written like this.

```
    not_gate ca cb /\ not_gate cc cc /\
```

The second "not" gate's input and output are connected. If we use the traditional correctness statement and after simplification, we must prove the following (many parts are elided for clarity).

$$\ldots \wedge (cc = \,^{\sim}cc) \wedge \ldots \Longrightarrow \ldots \text{specification} \ldots$$

The left hand side of the implication should reduce to false, so the theorem may be proved. Supposedly then the model *does* implement the specification, even though there is an error in the model of the implementation.

We follow the lead of other areas of computer science and require two different specifications: a "liveness" specification and a "safety" specification. The more cautious correctness statement we propose is this.

$$D_{liveness\ spec} \Longrightarrow D_{impl} \wedge D_{impl} \Longrightarrow D_{safety\ spec}$$

For the decoder example, we want any implementation to select one of the outputs for each valid address. We can formalize the liveness specification as follows.

```
decode_spec_live a0 a1 r0 r1 r2 =
  decode_constraint a0 a1 ==> r0 \/ r1 \/ r2
```

Here is the corresponding correctness statement. The predicate decode_spec_safety is just decode_spec from earlier.

```
decode_spec_live a0 a1 r0 r1 r2
                     ==> decode_impl a0 a1 r0 r1 r2
    /\ decode_impl a0 a1 r0 r1 r2
                     ==> decode_spec_safety a0 a1 r0 r1 r2
```

This correctness statement should prevent the incorrect implementation model from causing a falsely confirming proof.

3 Does it Matter?

Several people commented that even this correctness statement is not absolutely dependable. For example, the safety specification may be trivially true or the liveness specification may be contradictory. In either case, the statements may be verified even though there is an error. What can be done to strengthen the result?

To examine for a clause for falsity, one can try to prove the converse, for example,

```
~(decode_spec_live a0 a1 r0 r1 r2)
```

Alternatively, finding even one set of arguments which satisfies the predicate (makes it evaluate to true) means the predicate is not a contradiction. Validation can also help support (or disprove) that the specification means what is intended.

Examining a clause for trivial truth is equivalent to trying to prove, for instance,

```
decode_spec_safety a0 a1 r0 r1 r2
```

If this can be proved to be a theorem, it is clearly too weak: it specifies that *any* behavior is safe. One can evaluate specific combinations of input which should be unsafe, that is, for which the predicate should be false.

Contradictory implementations don't seem to have been a problem. Why not? We suggest some possible reasons.

- Most implementation models are structured so inadvertently writing a contradiction is unlikely.
- An incorrect implementation (or specification) usually results in a failure to prove, rarely a proof of an incorrectly formed statement.
- During proofs one notices that the proof was too easy, for instance, witnesses for existentially quantified variables are never needed.
- Finally proofs are typically used after the models are tested or simulated, so such gross errors are unlikely.

Proving the implementation does something via a liveness specification is akin to proving termination for software semantics: it is not central to most checking, but may uncover some errors. It seems prudent to consider the possibility of a false positive result.

4 Conclusion

We propose that implementations must be shown to satisfy some liveness conditions, that is, not be contradictions, in addition to implementing allowed behaviors. Although it has not been found to be a problem, this two-part correctness statement should add more assurance that when the goal is proved, the design is correct.

References

[1] Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A brief introduction to formal methods. In *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference (CICC '96)*, pages 377–380. IEEE, 1996.

[2] Solange Coupet-Grimal and Line Jakubiec. Coq and hardware verification: A case study. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, August 1996.

[3] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[4] Thomas F. Melham. Abstraction mechanisms for hardware verification. In G. M. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis, Proceedings of the Workshop on Hardware Verification*, January 1987.