ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

Mary Sheeran, Chalmers Technical University, Sweden, and Satnam Singh, University of Glasgow, UK (Eds)

Designing Correct Circuits

Proceedings of the 3rd Workshop on Designing Correct Circuits (DCC96), Båstad, Sweden, 2-4 September 1996

ISBN: 3-540-76102-0

Paper:

Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit

Paul S. Miner and Steven D. Johnson

Published in collaboration with the British Computer Society

BCS

Springer

Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit

Paul S. Miner
Flight Electronics Technology Division, NASA Langley Research Center
Hampton, VA, USA

Steven D. Johnson
Department of Computer Science, Indiana University
Bloomington, IN, USA

Abstract

In previous work, we explored the interaction between different formal hardware development techniques in the implementation of a fault-tolerant clock synchronization circuit. This case study presents a clever optimization of the earlier design and illustrates how we have extended our framework to support its incremental design refinement. The primary design tool represents circuits as systems of stream equations, where each stream corresponds to a signal within the circuit. These signals are annotated with invariants which can be established using proof by co-induction. These invariants are exploited to verify localized design refinements. This study lays groundwork for a more formal integration of disparate reasoning tools.

1 Introduction

A significant amount of effort within the formal methods community has been focused on how to verify hardware using particular verification systems. Not as much time has been spent asking what type of reasoning support we need to formalize the design process. The case study presented here is subject to the same criticism. The exercise began as an attempt to derive a clock synchronization circuit using the Digital Design Derivation (DDD [1]) system. When it became clear that DDD could not handle all the development steps, the effort evolved into exploring how a mechanical theorem proving system (PVS [8]) could support the derivation process. Lost in the effort was the question of how to best use formal techniques in the hardware development process. Much of this process is tedious and does not require sophisticated reasoning support. Our mission as researchers in this field should be to identify the types of reasoning that are useful to this process, and to develop those. These efforts toward developing a verified synchronization circuit have led to the position that a formal design process should be as simple as possible, but should allow sufficient flexibility to the designer to make aggressive optimizations to the design.

Early research in applied formal methods was concerned with basic questions of design and implementation correctness. We are now beginning to explore how formal reasoning interacts with intelligent design processes. This case study addresses a clever refinement to a design that is already well along the path toward realization. The goal is to preserve or reestablish the implementation's correctness while sustaining a secure verification path. The cleverness of the refinement lies in its exploitation of undisclosed properties of both the implementation and the environment in which it is to operate. Consequently, the formal characterizations of both the design and the implementation are extended.

Effective automated support of formal methods must accommodate multiple distinct modes of reasoning. One motive of this study is to explore heterogeneous reasoning and contribute to the growing experience with it. Derivation based formalisms—reasoning systems that employ transformations rather than logical inference—are relatively effective for routine design refinement. However, because they are dedicated to preserving specific refinement relations, they are not as general as deduction based systems, where the implementation relation can be expressed within the formalism. The developers of the DDD system were confronted with this limitation in contrasting a formal derivation of the FM8502 [4] and FM9001 [1, 2] microprocessors with Hunt's proofs of correctness in the theorem prover *Nqthm*.

In particular, Hunt's implementation of a functional memory model by an explicitly synchronized process exposed a gap in the derivation path. While this particular kind of problem has been addressed [17, 10], we believe that derivation gaps are an inevitable consequence of creativity in design and engineering.

On the other hand, generality hardly justifies the use of a theorem prover for all verification tasks. Even if one has somehow incorporated automatic provers and rewriters for lower level tasks, we believe that reasoning environments should support a variety of reasoning formalisms. At some point, such a system may employ the more unified view of a logical framework, but, for the present, experience is needed in the coordinated use of multiple interactive systems.

The technique presented here to augment DDD style derivation with PVS theorem proving support is not restricted to either DDD or PVS. Essentially, we present an effective means to establish invariants on signals within a circuit so that we can verify context-dependent optimizations of a circuit design. These invariants (assertions on signals) are established using co-induction, as is the verification of the optimization. Our goal is to develop a formalized design environment that supports annotation of signals with invariants (and handles all the associated bookkeeping aspects), so that a designer can explore various optimizations in a rigorous manner.

2 Related Work and Prior Developments

In the study described here, the DDD system provides mechanized support for behavioral and structural transformations, while PVS supports theorem proving activities. The interaction between these systems requires manual support at present, although we are laying the groundwork for mechanizing it. We view DDD and PVS as examples of autonomous reasoning peers. They are each useful tools in a formal hardware development process.

A case study by O'Leary, Leeser, Hickey and Aagaard, outlining the verification of a binary non-restoring square root implementation, reflects a contrasting perspective on heterogeneous reasoning [3]. The design and its proof of correctness develop through several stages of program transformation before a structural description emerges. These structures are then refined in several stages toward a realizable hardware description. Their study, like ours, exhibits both derivational and deductive reasoning processes, but within the unified framework of *Nuprl*. We believe the authors would argue in favor of such a framework as a prerequisite for heterogeneous reasoning.

Miner presents a verified class of fault-tolerant clock synchronization algorithms and an informal sketch of a hardware realization [5]. The general algorithm was verified using the mechanized proof system EHDM [11]. A hardware realization of the verified algorithm was developed and tested, but the hardware was not formally verified with respect to the algorithm [6, 14]. In [7], a core circuit design was developed using a variety of formal reasoning systems. The circuit was developed using a combination of the Prototype Verification System (PVS) developed at SRI [8], the DDD system developed at Indiana University [1], and a BDD-based tautology checker. This formal development identified a small improvement over the original design. While that work was in progress, Torres-Pomales identified a more significant improvement [15]. This paper explores how Torres-Pomales' optimization can be incorporated into the formal design framework proposed in [7]. This optimization trades space for time in a clever manner and led us to the conclusion that a formal design environment should be flexible enough to accommodate engineering insight. The argument justifying the optimization requires arithmetic reasoning including integer division, so its justification requires more than simple transformational techniques and also places the optimization outside the realm of model-checking approaches.

Specifically, we look at how the optimization can be incorporated with minimal impact on the surrounding proof. In essence, we want a transformation rule in DDD that allows a subsystem to be replaced by a behaviorally equivalent variant as established in PVS. To accomplish this, we need the means to transfer expressions between DDD and PVS, a theory of streams built into PVS, and a mechanism for sanctioning ad hoc transformations in DDD.

Since the goal of our verification activities is developing working hardware, a VLSI implementation of the formally developed circuit design has been fabricated and tested. The circuit layout was manually generated using conventional design tools, so the link between the fabricated circuit and the design is not completely formal. The VLSI realization has been incorporated into the full fault-tolerant clock synchronization system described by Torres-Pomales [15]. The circuit worked perfectly on all tests.

3 Verification Strategy

In order to carry out the verification reported here, we needed to identify what role each formal system would play in the development process. Ideally, the hardware development should not depend directly upon a general purpose proof system. Most theorem proving systems require a great deal of experience before they can be used effectively. However, such a system is indispensable in exploring the reasoning required for a formalized design process.

The principle design tool is DDD, and we wish to use it in a manner to minimize the application of general purpose verification activities. For the effort reported here, we used a shallow embedding of DDD's representation of hardware in PVS. Our primary motivation was justification of custom refinements, not in reasoning about the DDD approach to hardware design.

Figure 1 depicts our view of the design hierarchy and illustrates where each tool contributes to the design process. At the top-most level are mathematical properties that the resulting design must satisfy. A general purpose mechanical theorem proving system supports the verification of algorithms that ensure these requirements. The verified algorithm should be as general as is reasonably possible, so that the design space is not unduly restricted. The verified algorithm is (manually) translated to a DDD specification. This specification is refined within DDD and then transformed into an architecture. A sequence of basic DDD transformations, augmented with localized PVS verifications, determines the final architecture. At the final stage, DDD maps the abstract representations of data into boolean representations. Additional transformations may be applied prior to using conventional design tools to produce a physical realization of the design.

3.1 Overview of DDD

DDD implements a formal design algebra for developing correct digital circuit descriptions. The designer interactively transforms high level behavioral specifications into a description suitable for entry into hardware synthesis tools. The top level describes the intended behavior of the circuit using a collection of mutually recursive function definitions in tail-form. Each function corresponds to a control state and arguments to the functions represent the visible storage elements in the design. Transformations at this level allow the designer to modify the control structure of an architecture while preserving the functional correctness, relative to synchronization constraints. Once the control structure is determined, DDD automatically transforms the behavioral specification into an initial architectural level description.

DDD represents the structure of a digital system using a system of mutually recursive stream equations. A stream in DDD is an infinite sequence of uniformly typed values,

$$X = [x_0, x_1, x_2, \ldots]$$

The stream constructor '`cs`' adds an element to the front of the sequence

$$\texttt{cs}(z, X) = [z, x_0, x_1, x_2, \ldots]$$

Function '`cs`' models a delay element with initial value z. Functions extend to sequences so that $f(X, Y)$ denotes $[f(x_0, y_0), f(x_1, y_1), \ldots]$. DDD uses a system of equations to define a network of streams; recursion in such systems represents feedback in the circuit. For example, the following stream equation defines a loadable counter circuit.

$$\mathit{COUNT} = \texttt{cs}(i, \mathit{MUX}(S, L, \mathit{INC}(\mathit{COUNT}))) \tag{1}$$

where i is the initial integer value of the counter, *INC* and *MUX* are the increment and selection functions lifted to streams, and streams S and L are the multiplexor select signal and load input respectively. Within DDD, free variables in a system of stream equations are bound by a system level abstraction. This abstraction defines the input signals for the circuit. The circuit's output is a subset of the named streams in the system of equations.

3.2 Overview of PVS

PVS is a general purpose verification system developed at SRI International [8]. It consists of an expressive specification language coupled with a powerful mechanical theorem prover. The PVS specification language is based on higher-order logic. The base types include the booleans and the real numbers. The language includes predicate sub-types. The other numeric types are defined as sub-types of the reals. One consequence of introducing predicate sub-types is that the resulting type system is undecidable. Thus, PVS automatically generates proof obligations called type-correctness conditions (TCCs) when it type-checks a theory. Theories can be parameterized, providing some support for parametric polymorphism. PVS also allows for dependent types and several standard computer science type constructors such as records, tuples, and lists. PVS includes a prelude theory that defines a large collection of useful results. User defined libraries provide a mechanism to extend PVS with domain-specific theories.

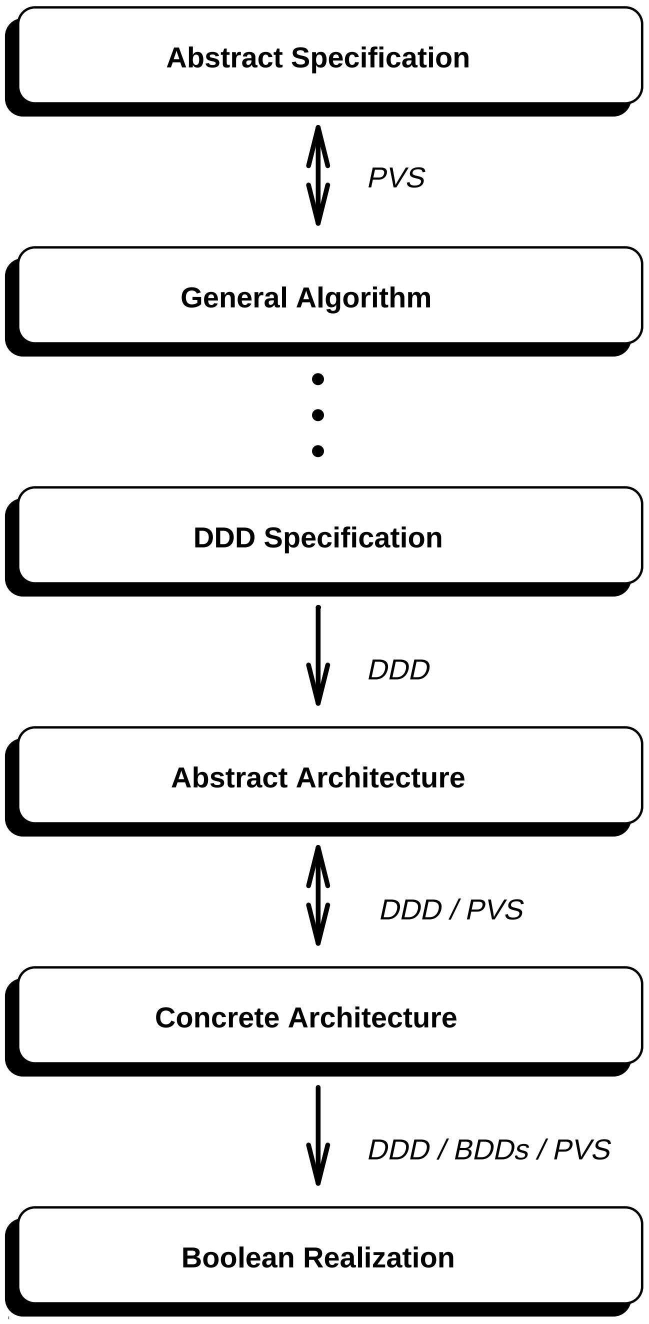

Figure 1: Abstract View of Design Hierarchy

PVS provides an interactive theorem proving environment using a sequent calculus presentation of the proof goals. The prover includes decision procedures for ground linear arithmetic and equality. There is a strategy language similar to LCF-style tactics. Thus, the user can define high-level proof procedures. There are several powerful strategies distributed with PVS that automatically verify a large number of results. PVS allows the user to prove lemmas in any order. It maintains proof dependency analysis to ensure that all obligations have been discharged. Included in the analysis is an enumeration of all axioms used by the proof chain.

4 Reasoning about Streams in PVS

Recursive stream definitions are not directly supported by PVS. It was necessary to identify a mechanism to allow such objects to be defined in PVS. Although streams over type α can be represented as functions from natural numbers to α, this representation does not lend itself to direct definition by stream equations. The equational style of definition illustrates that streams can be viewed as a *co-inductive type*. Just as inductively generated types give rise to recursive function definitions and proofs by induction, co-inductive types allow for definition by co-recursion and proofs by co-induction [9]. Co-induction is a categorical dual of induction; induction principles are justified using least fixed point arguments, co-induction principles are justified using greatest fixed point arguments. Although the underlying formal basis of co-induction is an interesting area of study, our work is primarily concerned with the application of these techniques to hardware verification.

4.1 Stream Definition

Streams in PVS are defined as parameterized uninterpreted types constrained by a set of axioms. For X, Y, S of type `Stream[`α`]`, and $a : \alpha$, the following axioms hold:

```
Stream_cs_eta: AXIOM cs(hd(S), tl(S)) = S
Stream_hd_cs:  AXIOM hd(cs(a, S)) = a
Stream_tl_cs:  AXIOM tl(cs(a, S)) = S
Stream_eq:     AXIOM (X = Y) <=> (FORALL n: nth(X, n) = nth(Y, n))
```

For $f : \alpha \to \beta$, $g : \alpha \to \alpha$, and $a : \alpha$, define function $\texttt{corec}_{(f,g)} : \alpha \to \texttt{Stream}[\beta]$ using the following axiom:

```
corec_def:     AXIOM corec(f, g)(a) = cs(f(a), corec(f, g)(g(a)))
```

The standard representation of streams, $\mathbf{N} \to \alpha$, augmented with explicit function definitions for `cs`, `hd`, `tl`, `nth`, and `corec`, provides a model for these axioms.

Function `corec` provides us with enough machinery to define any DDD stream equation. For example, one possible PVS declaration defining a stream function that satisfies equation (3.1) is

```
COUNT((S : Stream[boolean]),
      (L : Stream[integer]),
      (i : integer)): Stream[integer] =
  corec(LAMBDA S, L, i : i,
        LAMBDA S, L, i : (tl(S), tl(L), mux(hd(S),hd(L),inc(i)))
       )(S, L, i)
```

In this definition, type α is instantiated using the tuple type

```
[Stream[boolean], Stream[integer], integer]
```

and β is instantiated with type `integer`. The following two facts are easily proven about `COUNT`.

```
hd_COUNT: LEMMA
  hd(COUNT(S, L, i)) = i

tl_COUNT: LEMMA
  tl(COUNT(S, L, i)) =
    COUNT(tl(S), tl(L), mux(hd(S), hd(L), inc(i)))
```

The proofs consist of expanding the definition of `COUNT` followed by rewriting with the stream axioms. To simplify subsequent proofs, we adopt the convention that for every stream defined in PVS, we introduce lemmas for simplifying the `hd` and `tl`. The next section introduces a proof principle that simplifies proofs of stream equality.

4.2 Stream Equivalence

Definition of streams using co-recursion enables a useful technique for proving two streams equal. A stream bi-simulation R is a sub-relation of the equality relation such that for any two streams x and y, if $x\ R\ y$ then $\mathtt{hd}(x) = \mathtt{hd}(y)$ and $\mathtt{tl}(x)\ R\ \mathtt{tl}(y)$. The PVS type declaration defining the type of bi-simulations between streams over α is:

```
Bisimulation: TYPE =
  {R: PRED[[Stream[alpha], Stream[alpha]]] |
      FORALL X, Y: R(X, Y) =>
                 hd(X) = hd(Y) & R(tl(X), tl(Y))}
```

PVS automatically generates proof obligations for any object declared to be of this type. The following theorem provides a tool for proving stream equivalence by exhibiting a suitable bi-simulation.

```
co_induct: THEOREM
  (EXISTS (R: Bisimulation): R(X, Y)) => X = Y
```

The PVS proof of theorem `co_induct` consists of rewriting with axiom `Stream_eq` followed by induction on the natural numbers. We can now use co-induction to prove (in PVS) that the co-recursive definition of `COUNT` satisfies equation (3.1).

```
COUNT_eqn: LEMMA
  FORALL (S : Stream[bool]), (L : Stream[int]), (i : int):
    COUNT(S, L, i) = cs(i, MUX(S, L, INC(COUNT(S, L, i))))
```

Proof: The proof consists of rewriting with theorem `co_induct`, constructing a simple relation between integer valued streams from the given goal, and then showing that the constructed relation is a bi-simulation. The relation R is

```
{(I, J : Stream[integer]) |
  EXISTS (S : Stream[bool]), (L : Stream[int]), (i : int):
    I = COUNT(S, L, i) &
    J = cs(i, MUX(S, L, INC(COUNT(S, L, i))))}
```

This relation is constructed using information from the original goal. For every `S`, `L`, and `i`, this relation contains the pair

```
( COUNT(S, L, i), cs(i, MUX(S, L, INC(COUNT(S, L, i)))) )
```

All that remains is to show that this relation satisfies the type constraints of a bi-simulation. Take an arbitrary pair that is in the relation

```
( COUNT(S', L', i'), cs(i', MUX(S', L', INC(COUNT(S', L', i')))) )
```

There are two cases to consider:

Heads: The PVS proof for this case consists of rewriting with lemma `hd_COUNT` and axiom `hd_cs`.

```
hd(COUNT(S', L', i')) = i' = hd(cs(i', MUX(S', L', INC(COUNT(S', L', i')))))
```

Tails: The goal is to show that the pair

```
( tl(COUNT(S', L', i')), tl(cs(i', MUX(S', L', INC(COUNT(S', L', i'))))) )
```

is contained in the relation R. The proof consists of rewriting these two streams with `tl_COUNT`, `tl_cs`, `MUX_def`, `hd_INC`, `tl_INC`, and `hd_COUNT`, producing the following equivalences.

```
tl(COUNT(S', L', i')) =  COUNT(tl(S'), tl(L'), mux(hd(S'), hd(L'), inc(i')))

tl(cs(i', MUX(S', L', INC(COUNT(S', L', i')))))
  = MUX(S', L', INC(COUNT(S', L', i')))
  = cs(mux(hd(S'), hd(L'), hd(INC(COUNT(S', L', i')))),
       MUX(tl(S'), tl(L'), tl(INC(COUNT(S', L', i')))))
  = cs(mux(hd(S'), hd(L'), inc(i')),
       MUX(tl(S'), tl(L'),
           INC(COUNT(tl(S'), tl(L'), mux(hd(S'), hd(L'), inc(i'))))))
```

By instantiating `S` with `tl(S')`, `L` with `tl(L')`, and `i` with `mux(hd(S'), hd(L'), inc(i'))`, we estatblish that the tails are in the relation. □

We have written a PVS strategy named `(CO-INDUCT-AND-SIMPLIFY)` that completely automates the above proof steps. This strategy suffices to automatically discharge several standard stream identities.

4.3 Signal Invariants

In order to justify some refinements, it is necessary to establish invariants on the input signals. The following PVS theory fragment defines predicate `Invariant` to be true for any boolean valued stream that is true at every finitely accessible point. At first glance, this appears to be a useless definition. However, when used in conjunction with PVS' dependent type mechanism it provides a useful means to define an invariant relating a collection of signals.

```
Invariant(A : Stream[bool]) : bool = (A = const(true))

Invariant_hd: LEMMA Invariant(A) => hd(A)

Invariant_tl: LEMMA Invariant(A) => Invariant(tl(A))

CoInductive_Assertion: TYPE =
  {P| forall A: P(A) => hd(A) & P(tl(A))}

co_induct: THEOREM
   (EXISTS (P: CoInductive_Assertion): P(A)) => Invariant(A)
```

An example of how predicate `Invariant` may be used, consider the following parameterized type declaration (A and R are boolean valued streams):

```
S(R): TYPE =
    {A| Invariant(IF R THEN NOT tl(A)
                         ELSE A => tl(A) ENDIF)}
```

A stream of type $S(R)$ corresponds to a signal that once asserted remains asserted, unless it is reset by boolean stream R.

Theorem `co_induct` provides a mechanism for proving that an arbitrary boolean valued streams is always true. To establish an invariant property about a stream, it is sufficient to show that that property is contained in some coinductive assertion. The PVS strategy `(CO-INDUCT-AND-SIMPLIFY)` automatically verifies many invariant properties.

5 Fault-Tolerant Clock Synchronization

In a fault-tolerant computer architecture, the clocks of the redundant computing elements need to be synchronized to ensure that they operate in a coordinated manner. The synchronization algorithm must also tolerate a bounded number of failures. The property that a synchronization algorithm must ensure is that:

For any two clocks C_p and C_q that are nonfaulty at time t

$$|C_p(t) - C_q(t)| \leq \delta$$

Clock synchronization algorithms are designed so that by periodically exchanging values of clocks and executing a fault-tolerant averaging function, the above property is guaranteed.

Schneider [12] demonstrates that many fault-tolerant clock synchronization algorithms can be treated as refinements of a general protocol. Shankar [13] and Miner [5] have provided mechanically checked proofs of Schneider's paradigm. Miner's verification is the top-level specification for the circuit developed here. A generalized view of the algorithm employed by each participant in the protocol is:

```
do forever {
    exchange clock values
    determine adjustment for this interval
    determine local time to apply correction
    when time, apply correction }
```

Schneider's paradigm is parameterized by

- N—the number of clocks participating in the protocol, $N > 0$
- F—the number of faults tolerated
- A mechanism for exchanging clock values. The relationship between N and F depends on this mechanism. Usually, $N > 3F$.

  We use θ to denote a collection of readings from clocks in the system. In the mechanically verified theory, θ is a function from clock indices to clock readings.
- R—the nominal duration of a synchronization interval.
- *cfn*—a convergence function that must satisfy three properties:

  Translation Invariance The function depends only on the relative magnitude of the readings, not the absolute magnitude.

  Precision Enhancement For any two good clocks with similar estimates of other clock's values, the result of computing the convergence function is similar.

  Accuracy Preservation If the readings from good clocks are sufficiently similar, then the computed value of the convergence function is close to all good clocks.

The fault-tolerant midpoint convergence function,

$$\mathit{cfn}_{MID}(\theta) \;=\; \left\lfloor \frac{\theta_{(F+1)} + \theta_{(N-F)}}{2} \right\rfloor, \text{ where } \theta_{(m)} = \text{ the } m\text{th largest value in collection } \theta$$

employed in the Welch and Lynch [16] clock synchronization algorithm, possesses the required properties of a convergence function [5].

In previous work [7], we developed a hardware realization of this verified algorithm using a combination of formal design techniques. The verified algorithm was manually translated into a DDD behavioral level specification. A standard technique was chosen for exchanging values between the redundant clocks. At a fixed offset into each synchronization interval, a signal is broadcast to the other participants in the protocol. The estimate for a remote clock's value is computed by determining the difference between the expected offset for receiving this signal and the actual offset when it is received. Using standard DDD transformations, an ad hoc refinement verified using PVS, and BDD-based tautology checking, we developed a hardware description suitable for realization using a field-programmable gate-array.

In a separate effort, Torres-Pomales [15] discovered a more efficient realization of the core synchronization circuit. We were faced with the problem of how to incorporate this optimization into our existing verification. We isolated the sub-circuit affected by the optimization, and then verified a localized refinement with respect to the existing design description.

5.1 Description of the sub-circuit

Figure 2 illustrates the core circuit computing the convergence function presented in [7]. The signal `RD` is the output of a counter. The signals `F1` and `NF` are boolean valued signals that indicate receipt of a synchronization signal from at least $F + 1$ and $N - F$ distinct participants in the protocol, respectively. Here, N represents the number of clocks in the protocol and F represents the number of physical faults tolerated by the system. By discarding readings from the F fastest and F slowest clocks, this protocol can tolerate F Byzantine failures.

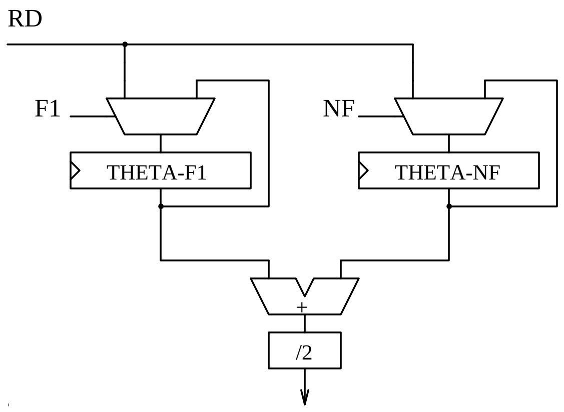

Figure 2: Core Circuit for Computing Convergence Function

The two registers capture the current value of the counter when the surrounding hardware receives signals from appropriately selected remote clocks. The verification discussed in [7] establishes that capturing just two readings in each synchronization interval is sufficient for correct execution of the algorithm. The proof technique employed for the ad hoc refinement in the earlier effort was not easily generalized. The difficulties encountered led us to refine our proof technique for verifying these custom transformations.

6 Optimization

Torres-Pomales discovered that the convergence function has a much more efficient realization [15]. He recognized that he could exploit the time interval between the $(F+1)$th and $(N-F)$th signals to partially compute the convergence function. His optimization consists of capturing the $(F+1)$th reading as before, but he then increments the captured value every other clock tick until the signal from the $(N-F)$th clock arrives. At this point the stored value is exactly the required value of the convergence function.

This next section will outline a technique to transform our previous design into Torres-Pomales' design. The optimized convergence function is depicted in Figure 3. This optimization requires assumptions about the input

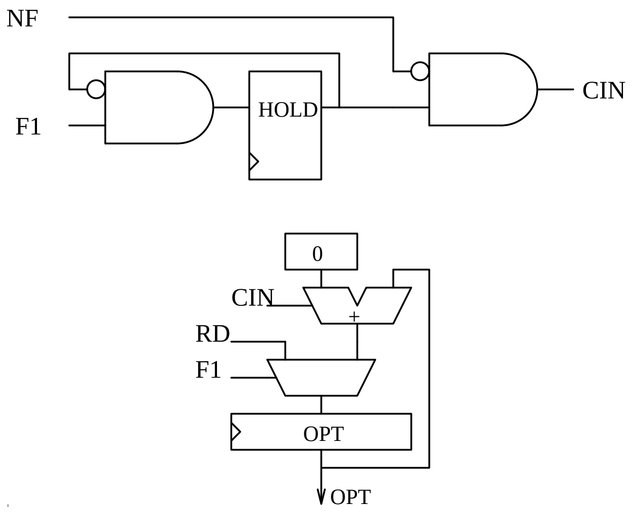

Figure 3: Optimized Convergence Function

signals. We already know that the integer stream RD is the output of a counter, so it increases by 1 each tick of the clock during the interval between resets. We can also show that the Boolean streams F1 and NF follow the behavior shown in Figure 4. The next section outlines a proof that this transformation is correct.

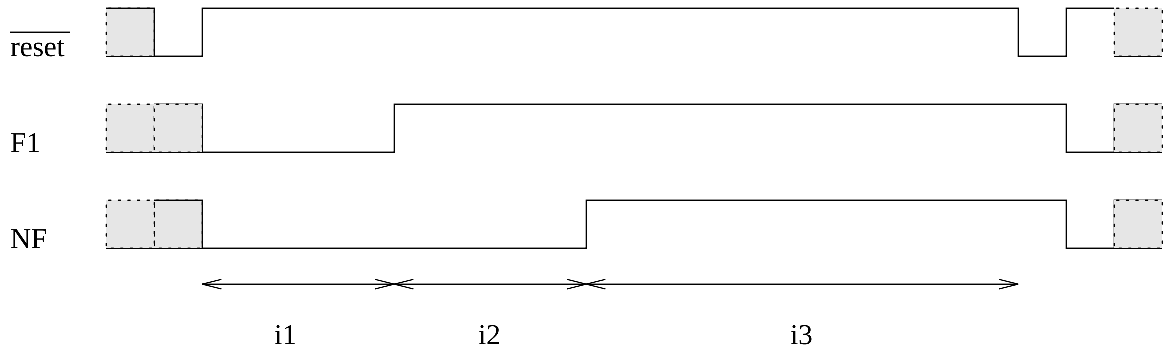

Figure 4: Signal Assumptions

6.1 Verification

The original circuit (Figure 2) is described by the following collection of stream equations:

$$\begin{aligned}
\texttt{THETA-F1} &= \texttt{cs}(i, \texttt{MUX(F1, RD, THETA-F1)}) \\
\texttt{THETA-NF} &= \texttt{cs}(i, \texttt{MUX(NF, RD, THETA-NF)}) \\
\texttt{CFN} &= \left\lfloor \frac{\texttt{THETA-F1} + \texttt{THETA-NF}}{2} \right\rfloor
\end{aligned}$$

The optimized circuit (Figure 3) is described by these stream equations:

$$\begin{aligned}
\texttt{HOLD} &= \texttt{cs(b, F1 \& }\neg\texttt{HOLD)} \\
\texttt{CIN} &= \texttt{HOLD \& }\neg\texttt{NF} \\
\texttt{OPT} &= \texttt{cs}(i, \texttt{MUX(F1, RD, INC(OPT,CIN))})
\end{aligned}$$

We wish to prove that OPT = CFN, given some assumptions about the input streams F1, NF, and RD. Streams CFN and OPT can be defined as functions from two Boolean streams, F1 and NF; an integer stream, RD; and initial values for the state—CFN has two integer-valued storage elements and OPT has one Boolean and one integer storage location. The following declarations define the necessary stream functions in PVS.

```
THETA(A,I,i): Stream[int] =
        corec(lambda A,I,i:i,
              lambda A,I,i:(tl(A),tl(I),mux(hd(A),hd(I),i)),
              (A,I,i))
CFN(A,B,I,i,j): Stream[int] = DIV2(THETA(A,I,i)+THETA(B,I,j))
HOLD(A,a): Stream[bool] =
        corec(lambda A,a:a,
              lambda A,a:(tl(A),hd(A) and not a),
              (A,a))
CIN(A,B): Stream[bool] = A AND NOT B
OPT(A,B,I,i): Stream[int] =
        corec(lambda A,C,I,i: i,
              lambda A,C,I,i: (tl(A),tl(C),tl(I),
                                mux(hd(A),hd(I),
                                inc_c(i,hd(C)))),
              (A,C,I,i))
```

Sub-circuits that involve feedback are defined using `corec`. The other functions are defined using composition of simple functions lifted to streams. Each of the defined stream expressions satisfies the corresponding stream equation from above, e.g. the equations

$$\begin{aligned}\texttt{THETA}(A,I,i) &= \texttt{cs}(i,\texttt{MUX}(A,I,\texttt{THETA}(A,I,i)))\\ \texttt{HOLD}(A,a) &= \texttt{cs}(a,\ A\ \texttt{\&}\ \neg\texttt{HOLD}(A,a))\\ \texttt{OPT}(A,C,I,i) &= \texttt{cs}(i,\ \texttt{MUX}(A,I,\texttt{INC}(\texttt{OPT}(A,C,I,i),C)))\end{aligned}$$

have been proven in PVS using `(CO-INDUCT-AND-SIMPLIFY)`. In addition, lemmas for simplifying both the head and tail of each stream function have been proven.

We need to prove that

```
CFN(F1, NF, RD, i, i) = OPT(F1,CIN(HOLD(F1,b),NF), RD, i)
```

given suitable assumptions about the parameters. Figure 4 illustrates the assumptions about `F1` and `NF` with respect to a (true low) reset signal. Signal `RD` is the output of a counter; in the absence of a reset, it increments by one each tick of the local clock. We define the following parameterized types in PVS.

```
S(R): TYPE =
    {A| Invariant(IF R THEN NOT tl(A)
                            ELSE A => tl(A) ENDIF)}

C(R): TYPE =
    {I| Invariant(NOT R => EQ(tl(I),INC(I)))}
```

$S(R)$ is the type of Boolean valued signals that will remain asserted, if ever asserted, until one tick after the next reset event signaled by R. Streams `F1` and `NF` are both of type $S(R)$. $C(R)$ defines the type of all streams that are outputs of a counter that may be reset (to some unspecified value) by R. Stream `RD` is of type $C(R)$. In addition to these invariants on the input streams, we must also constrain both the relationship between `F1` and `NF`. This is accomplished via the PVS dependent type mechanism. The correctness theorem is:

```
Optimize_correct: THEOREM
 FORALL (R  : Stream[boolean]),
        (RD : C(R)),
        (F1 : S(R) | NOT hd(F1)),
        (NF : S(R) | Invariant(NF => F1)),
        (i  : integer),
        (b  : boolean):
   CFN(F1, NF, RD, i, i) =
     OPT(F1, CIN(HOLD(F1, b), NF), RD, i)
```

In order to prove the optimization correct, it is sufficient to exhibit a bi-simulation between `CFN` and `OPT`.

A suitable bi-simulation, B, for proving theorem `Optimize_correct` is:

```
 {(I, J)|
   EXISTS (R  : Stream[boolean]),
          (RD : C(R)),
          (F1 : S(R)),
          (NF : S(R) | Invariant(NF => F1)),
          (i  : integer),
          (j  : integer | hd(F1) and not hd(NF) => (j + 1 = hd(RD))),
          (b  : boolean | hd(F1) and not hd(NF) => (b = odd?(i + j))):
     I = CFN(F1, NF, RD, i, j) &
     J = OPT(F1, CIN(HOLD(F1, b), NF), RD, floor((i + j) / 2))
  }
```

The definition of this bi-simulation makes use of the PVS dependent type mechanism in the declaration of existentially quantified variables. Streams `RD`, `F1`, and `NF` are all dependent on R. The type of Boolean stream `NF` is further constrained by `F1`. Stream `NF` cannot be asserted until after `F1` is asserted. These type constraints are the same as presented in the statement of Theorem `Optimize_correct`. Also, Boolean variable b is constrained to equal `odd?`$(i + j)$ whenever `hd(F1)` is asserted and `hd(NF)` is not. Under the same conditions, $j + 1 =$ `hd(RD)`. These restrictions on b and j in the bi-simulation essentially state invariants about `HOLD` and `THETA-NF` during the sub-interval $i2$ depicted in Figure 4. In addition, the current state of the optimized sub-circuit is functionally related to the current state of the original circuit. These invariant properties constitute the primary reason that this refinement is correct. All of these constraints are used in the following proof by co-induction.

Proof: (of `Optimize_correct`)

A co-inductive proof subdivides into two major cases. The first case consists of showing that the pair of streams to be proven equal are included in the candidate bi-simulation. The second case consists of showing that the given relation is indeed a bi-simulation.

The first step is to show

(`CFN(F1, NF, RD, i, i)`, `OPT(F1, CIN(HOLD(F1, b), NF), RD, i)`) $\in B$

This consists of proving the following goal:

```
     EXISTS (R  : Stream[boolean]),
            (RD : C(R)),
            (F1 : S(R)),
            (NF : S(R) | Invariant(NF => F1)),
            (i  : integer),
            (j  : integer | hd(F1) and not hd(NF) => (j + 1 = hd(RD))),
            (b  : boolean | hd(F1) and not hd(NF) => (b = odd?(i + j))):
       CFN(F1', NF', RD', i', i') = CFN(F1, NF, RD, i, j)
          &
       OPT(F1', CIN(HOLD(F1', b'), NF'), RD', i')
         = OPT(F1, CIN(HOLD(F1, b), NF), RD, floor((i + j) / 2))
```

We instantiate `R`, `RD`, `F1`, `NF`, `i`, `j`, and `b` with `R'`, `RD'`, `F1'`, `NF'`, `i'`, `i'`, and `b'` respectively. PVS strategy `(INST?)` automatically guesses these instantiations. The type constraint on stream `F1'` in the statement of the theorem ensures `NOT hd(F1')`, thus, there are no constraints on `j` or `b`. PVS decision procedures complete the proof. All that remains is to show that relation B is a bi-simulation. There are two cases:

Heads: Rewriting with `hd_CFN`, `hd_DIV2`, `hd_plus`, `hd_THETA`, and `hd_OPT` establishes for any $(X, Y) \in B$, `hd(`X`) = hd(`Y`) =` $\lfloor (i + j)/2 \rfloor$.

Tails: For any $(X, Y) \in B$, show `(tl(`X`),tl(`Y`))`$\in B$. Rewriting with all of the available lemmas about `hd` and `tl` simplify the tails of X and Y as follows:

```
   tl(CFN(F1', NF', RD', i', j'))
      = CFN(tl(F1'), tl(NF'), tl(RD'),
            mux(hd(F1'), hd(RD'), i'),
            mux(hd(NF'), hd(RD'), j'))

   tl(OPT(F1', CIN(HOLD(F1', b'), NF'), RD', floor((i' + j') / 2)))
      = OPT(tl(F1'),
            CIN(HOLD(tl(F1'), hd(F1') AND NOT b'), tl(NF')),
            tl(RD'),
             mux(hd(F1'), hd(RD'),
                 mux((b' AND NOT hd(NF')), floor((i' + j') / 2),
                     1 + floor((i' + j') / 2))))
```

To show that this pair of streams is in B, we instantiate the existentially quantified variables of B appropriately. The PVS strategy `(INST?)` guesses incorrectly here. Appropriate instantiations are determined from the right-hand side of the equations above. Streams `tl(R')`. `tl(RD')`. `tl(F1')`, and `tl(NF')` instantiate `R`. `RD`. `F1`, and `NF`, respectively. The expressions “`mux(hd(F1'), hd(RD'), i')`” and “`mux(hd(NF'), hd(RD'), j')`” instantiate integers `i` and `j`. Expression “`hd(F1') AND NOT b'`” instantiates boolean variable `b`.

The main branch of the proof reduces to showing

```
   mux(hd(F1'), hd(RD'),
       mux((b' AND NOT hd(NF')), floor((i' + j') / 2),
           1 + floor((i' + j') / 2)))
           =
    floor((mux(hd(F1'), hd(RD'), i')
            + mux(hd(NF'), hd(RD'), j'))
                       / 2)
```

The PVS proof consists of bringing the type constraints on `NF'`, `j'` and `b'` into the sequent, simplifying the resulting expressions, and then applying PVS' brute-force strategy `(GRIND)`.

The rest of the proof involves satisfying the type correctness conditions generated by PVS when we instantiated the variables constrained by dependent type declarations. The correctness conditions for `RD`, `F1`, and `NF` follow from the fact that these streams satisfy an invariant. The correctness of the instantiations for b and j depends on the fact that `RD` is the output of a counter and that `F1` is asserted before (or simultaneously with) `NF`. □

The above argument may seem difficult. However, the only real difficulty lies in determining the appropriate invariants for the input streams and state variables. Once these are correctly chosen, the proof of stream equality by exhibiting a bi-simulation is mostly mechanical. This approach to verifying a local replacement forced us to focus directly on the mathematical justification for the replacement. The routine aspects of the verification are discharged in a mechanical fashion.

7 Establishing Invariants

The verification presented above is only valid if the input signals satisfy the corresponding invariants. These can also be established using a co-inductive proof. The signal `RD` is generated by the sub-circuit shown in Figure 5. This behavior of this circuit is described by the stream equation:

```
    LC = cs(i,
            SELECT(STATUS, INC(LC), INC(LC), INC(LC),
                           CLR(LC), INC(LC)))
```

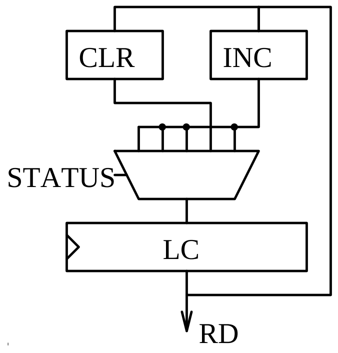

Figure 5: Counter Sub-Circuit

This is a variation of the counter circuit defined by equation (3.1). It is defined in PVS using co-recursion. We use the PVS judgement mechanism to assert that `LC` satisfies the type required by the verification of the optimization.

```
  C(R): TYPE = {I| Invariant(NOT R => EQ(tl(I),INC(I)))}

  JUDGEMENT LC HAS_TYPE [S:Stream[status_type],int->C(RST(S))]
```

The type judgement generates the PVS proof obligation:

```
   FORALL (S:Stream[status_type], i:int):
      Invariant(NOT RST(S) => EQ(tl(LC(S,i)),INC(LC(S,i))))
```

This is proven automatically using the PVS strategy `(CO-INDUCT-AND-SIMPLIFY)`. The requirements on `F1` and `NF` are discharged in a similar manner.

Since these proof obligations are often discharged automatically, it would be more productive to add a function to the derivational system to attempt a simple co-inductive proof prior to generating the necessary PVS theories. There is no need to use the power of a general purpose prover when a simple function added to the design tool can provide the same level of assurance.

8 Concluding Remarks

Optimizations of hardware designs often exploit implicit properties of the surrounding system. Approaches for formal hardware development need to include an effective means to represent and reason about changes in an evolving hardware design. Derivation-based formalisms provide a suitable framework for managing the routine design refinements, but cannot be expected to cover the possible design space. General purpose theorem proving systems, on the other hand, provide sufficient generality to capture arbitrary design refinements, but can be cumbersome for the more routine aspects of design. Formal design environments need to strike a balance between the two extremes.

In this paper, we presented a scenario where the generality of a general purpose proof system is necessary to complete the verification. However, the verification effort also identified a plausible extension to a derivational reasoning system. It is possible that a trivial bi-simulation is sufficient to justify an ad hoc refinement. In this case, it is not necessary to use the full power of a mechanized proof system. A simple extension to the derivation system can automatically attempt a trivial co-inductive proof. If it succeeds, the refinement is allowed, otherwise the necessary proof obligation is generated. In cases of design modifications resulting from clever engineering insight, the verification strategy presented here focuses effort on the mathematical justification for the refinement. The mundane aspects of the verification are handled automatically.

Acknowledgments

Bob Burger participated in the development and testing of the VLSI realization of the above circuit and also developed an Oct tools script that generates a layout from a functional description that is equivalent to the above circuit description. Wilfredo Torres-Pomales was instrumental in incorporating the circuit described here into the clock synchronization system. Jean-Yves Marion provided invaluable support during the initial development of the PVS Streams library.

References

[1] Bhaskar Bose. *DDD-FM9001: Derivation of a verified microprocessor*. PhD thesis, Computer Science Department, Indiana University, USA, 1994.

[2] Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation. In *Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods*. Springer, 1993.

[3] J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 52–71, Bad Herrenalb, Germany, September 1994. Springer Verlag. published 1995.

[4] Steven D. Johnson, R.M. Wehrmeister, and B. Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 385–404. Elsevier, 1989. IMEC 1989.

[5] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. Technical Paper 3349, NASA, Langley Research Center, Hampton, VA, November 1993.

[6] Paul S. Miner, Peter A. Padilla, and Wilfredo Torres. A provably correct design of a fault-tolerant clock synchronization circuit. In *Proceedings 11th Digital Avionics Systems Conference*, pages 341–346, Seattle, WA, October 1992.

[7] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *Proceedings 13th Symposium on Reliable Distributed Systems*, pages 128–137, Dana Point, CA, October 1994.

[8] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[9] Lawrence C. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report 304, University of Cambridge Computer Laboratory, July 1993.

[10] Kamlesh Rath. *Sequential-System Factorization*. PhD thesis, Computer Science Department, Indiana University, USA, 1995.

[11] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[12] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[13] Natarajan Shankar. Mechanical verification of a generalized protocol for byzantine fault-tolerant clock synchronization. In *Second International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236. Springer-Verlag, January 1992.

[14] Wilfredo Torres-Pomales. A hardware implementation of a provably correct design of a fault-tolerant clock synchronization circuit. Technical Memorandum 109001, NASA, Langley Research Center, Hampton, VA, July 1993.

[15] Wilfredo Torres-Pomales. An optimized implementation of a fault-tolerant clock synchronization circuit. Technical Memorandum 109176, NASA, Langley Research Center, Hampton, VA, February 1995.

[16] J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

[17] Zheng Zhu. *Structured Hardware Design Transformations*. PhD thesis, Computer Science Department, Indiana University, USA, 1992.