TECHNICAL REPORT NO. 405

Interaction of Formal Design Systems in the Development of a Fault-Tolerant Clock Synchronization Circuit[1]

Paul S. Miner[2]   Shyamsundar Pullela
Steven D. Johnson

Indiana University
Computer Science Department
Bloomington, IN 47405-4101

April 1994

[1]Research reported herein was supported, in part, by the National Science Foundation under grant MIP92-08745.
[2]On leave from NASA Langley Research Center, Hampton, VA 23681

Abstract

In this paper we propose a design strategy that exploits the strengths of different formal approaches to establish a reliable path from a mechanically verified high-level description to a concrete gate-level realization. We demonstrate the use of this approach in the realization of a fault-tolerant clock synchronization circuit.

We used the Digital Design Derivation system (DDD) to derive major portion of the design leaving relatively small portions to be verified either by use of a mechanical theorem prover (PVS) or by demonstrating boolean equivalence using Ordered Binary Decision Diagrams. DDD allows the designer to isolate areas of the design space where mechanized proof support can be most effectively applied, while maintaining the overall integrity of the development process. The interface between the different systems has not yet been completely formalized but we believe that our approach will provide an effective design path from high-level specifications to concrete realizations.

1 Introduction

Architectural components realizing fault-tolerant algorithms require effective design techniques. Even if the algorithm is formally verified, a simple bug in the realization could introduce a single point failure. There are a number of cases of mechanically verified fault-tolerant algorithms [11, 20, 12], as well as verified specifications of high-level system descriptions [6, 7, 16, 23]. In addition, there have been exercises demonstrating the application of mechanical theorem provers to the verification of hardware components realizing fault-tolerant algorithms [22, 1, 13]. The difficulty with these verified components is that each proof involves simplifying assumptions concerning the rest of the architecture. In particular, they assume that the redundant computing elements are operating in lock-step synchrony. While it is possible to achieve this using existing clock synchronization algorithms, Byzantine agreement can be achieved with looser synchronization. Furthermore, these verifications are not robust in the face of changes. That is, a small change in the design may invalidate much of the correctness proof.

A different approach is formal *derivation* of hardware. The Digital Design Derivation system (DDD) [2] implements a collection of behavior preserving transformations that allow a designer to realize a design that is "correct-by-construction". There have been a number of significant designs realized using DDD. In particular, Bose has formally derived the DDD-FM9001 [3] processor from Hunt's formal specification of the mechanically verified FM9001 microprocessor [8]. Another significant derivation is the development of a Scheme Machine, which consists of a garbage-collecting memory system and a CPU that executes high-level Scheme primitives [24]. During the above projects, it was recognized that a combination of DDD with mechanized proof support would be superior to either approach individually [10].

An important characteristic of DDD is that it allows the designer flexibility to explore the design space. It is fairly easy to explore a number of design possibilities in a formal manner, without committing to a particular design decision. In contrast, if a design is verified with a mechanical theorem proving system, it is unlikely that any of it will be altered, due to the difficulties inherent in modifying the correctness proof.

Conversely, there are some design optimizations that may not be effectively realized within a transformational system. In particular, some transformations are only applicable within the context of a certain design. Such transformations cannot be justified on general principles. We adopt the terminology of Saxe, et al. [18] and refer to these as an *ad hoc* transformations. Justification of such transformations requires theorem proving support. Thus, we would like to augment the derivational approach with mechanical theorem proving support thereby enabling the effective formal development of more efficient hardware.

This paper documents the first attempt to derive hardware from a parameterized formal specification of a fault-tolerant clock synchronization algorithm. This effort involves three different classes of formal design techniques and spans the spectrum from abstract properties to a realizable circuit design. In addition, each stage in the development maintains some generality, so the resulting circuit can be used in a variety of systems.

2 Fault-Tolerant Clock Synchronization

A critical function in a fault-tolerant architecture is synchronizing the clocks of the redundant computing elements. Schneider [19] demonstrates that many fault-tolerant clock synchronization algorithms can be treated as refinements of a general protocol. Shankar [20] and Miner [12] have provided mechanically checked proofs of Schneider's paradigm.

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 upon 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.

- *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 clocks 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.

- R—the nominal duration of a synchronization interval.

The fault-tolerant midpoint convergence function,

$$\begin{aligned} cfn_{MID}(\theta) &= \left\lfloor \frac{\theta_{(F+1)} + \theta_{(N-F)}}{2} \right\rfloor, \\ \text{where} \qquad \theta_{(m)} &= \text{ the } m\text{th largest value of } \theta \end{aligned}$$

employed in the Welch and Lynch [25] clock synchronization algorithm, possesses the required properties of a convergence function. A machine checked proof of this is presented by Miner [12]. In this paper, we will use the assumptions of Miner's verification as a top-level specification for a clock synchronization circuit employing the fault-tolerant midpoint convergence function.

3 Overview of Verification Procedure

This verification effort involves the use of three distinct classes of verification tools: mechanical theorem proving systems (EHDM [17], PVS [14]), a formal Digital Design Derivation system (DDD [2]), and automatic tautology checking using Ordered Binary Decision Diagrams (OBDDs [5]).

Figure 1 gives an overview of the various formal verifications performed. The top two levels in the figure represent Miner's mechanical verification of Schneider's paradigm [12]. This is a fully mechanized proof that any system that satisfies a small collection of properties is guaranteed to provide fault-tolerant clock synchronization. The work reported in this paper begins with a state-machine that represents a generalized clock synchronization algorithm. We claim without proof that this representation satisfies the appropriate assumptions of the EHDM verification.

This state machine is transformed into a structural description using DDD. This structural description is transformed into an architecture using both built-in DDD transformations and an *ad hoc* [18] transformation that requires external justification. Appropriate subsets of the DDD description are manually translated into the PVS logic. The PVS prover is then used to demonstrate that a particular substitution preserves behavior within the context of the rest of the design. Additional structural transformations are applied to yield a concrete architecture. This architecture is then transformed into a gate-level description using standard DDD projection functions, some custom projections justified by mechanized proof, and OBDDs to establish equivalence of projected modules to some custom components. This boolean level representation is targeted to a realization using the Actel Field Programmable Gate Array (FPGA) technology.

There are gaps in the formal development described here. The proof that the top-level DDD specification satisfies the appropriate requirements of the EHDM verification has not been done. Furthermore, the translations between DDD and PVS are manual; there is no formal interface defined between the two systems. Similarly, the translation to the source languge for the OBDDs is manual.

3.1 A brief introduction to DDD

(*The following description of DDD is adapted from [4]*)

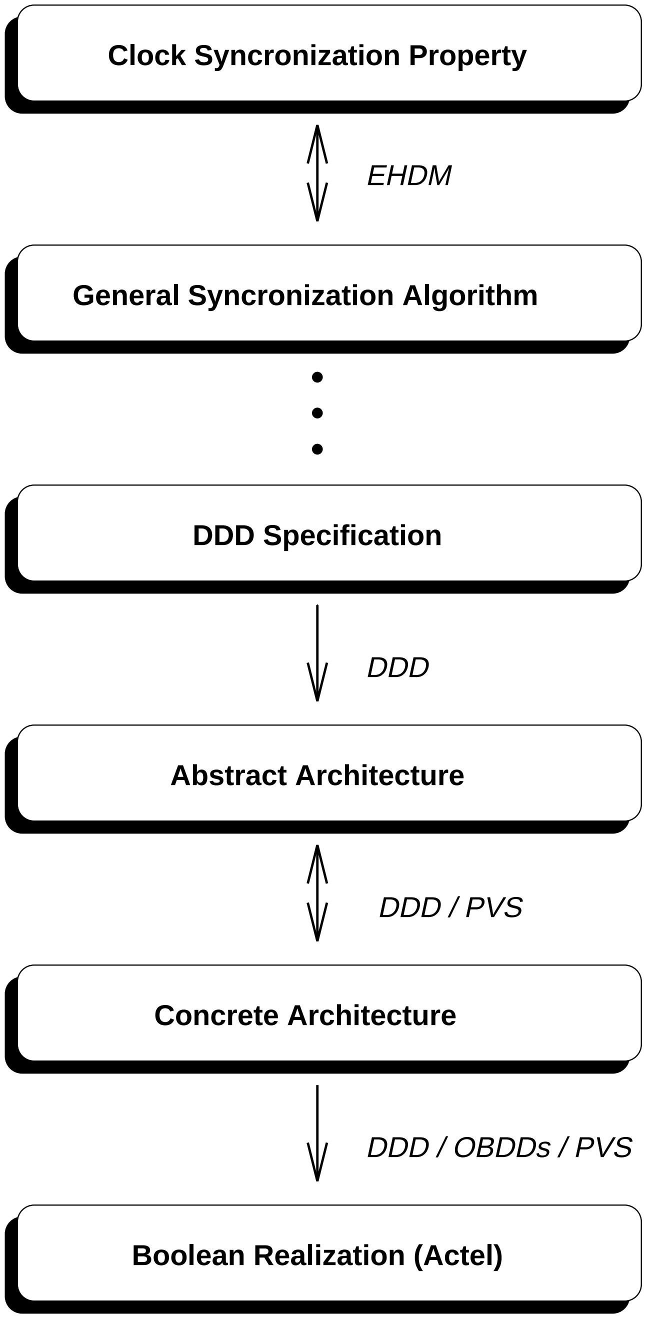

Figure 1: Verification Levels

DDD (Digital Design Derivation System) is a transformation system which implements a basic design algebra for synthesizing digital circuit descriptions from high level functional specifications [9, 2]. DDD is much like a theorem prover in the sense that it automates the transformations needed for circuit synthesis, but requires guidance to perform a derivation. DDD is implemented in the Lisp dialect Scheme as a collection of transformations that operate on s-expressions. The system is intended to provide a well founded, mechanized, algebraic tool set for design derivation; it is used interactively to transform higher level behavioral specifications into hierarchical boolean systems, to which logic synthesis tools are then applied.

In DDD, a sequence of transformations are applied to an initial specification defining a *derivation path* towards an implementation satisfying an intended set of design constraints. The initial behavior description is in a restricted class of *iterative* function-definition schemes. Behavior descriptions are folded and unfolded to achieve a proper scheduling of operations. From a suitable behavior description, DDD automatically builds an abstract *system* description composed of a decision combinator representing control, and a structural component representing architecture. A sequence of factorization steps decomposes this structural component into a communicating system of modules, encapsulating complex signals and functions as co-processes. A third class of transformations introduces a lower-level representation. Ultimately, this decomposition produces a hierarchy of boolean subsystems. Logic synthesis is used to assemble the subsystems to the appropriate technology.

4 Behavioral Specification

An abstract view of our behavioral specification is given in Figure 2 using the ASM chart notation from [15] . This corresponds to a top-level DDD specification. It establishes the control of the machine and identifies some of the registers, but leaves architectural components abstractly specified. In particular, we have not given an explicit representation for either θ or *cfn*. Our choice for *cfn* is the fault-tolerant midpoint convergence function

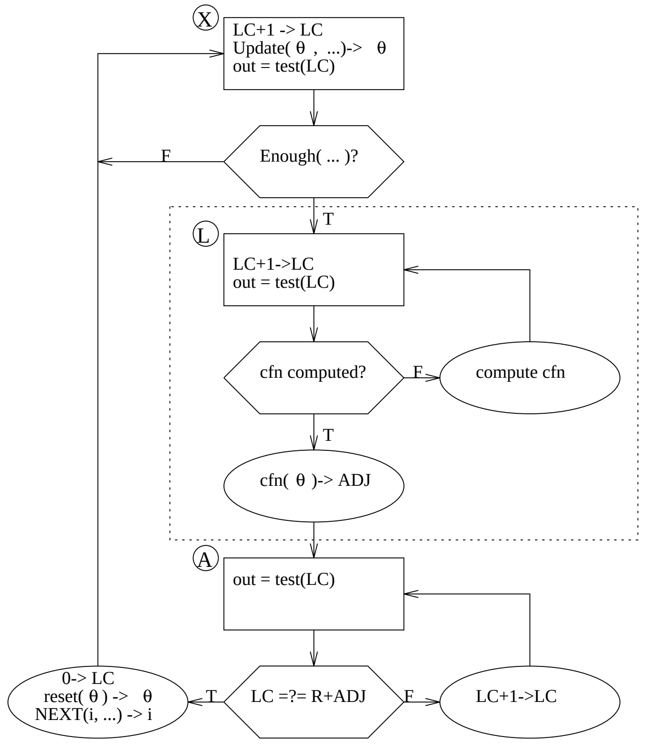

Figure 2: ASM Chart for General Algorithm

presented in section 2.

We will use a standard mechanism for exchanging readings. At a fixed offset into each synchronization interval, each participant in the protocol will transmit a signal. Upon receipt of that signal, each clock will capture its estimate for the transmitting clock. By analyzing delays in the communication mechanism we can arrive at an estimate of what the value of the local counter, LC, should be when a clock receives a signal from another clock that is perfectly synchronized. This is a constant value that we denote by Q. Thus, the value we capture when we receive a signal from a remote clock is $LC - Q$.

We now need specifications for the functions manipulating θ. Our arguments to Update include θ, a collection of N signals, and a reading, $LC - Q$. Since our convergence function chooses elements of θ based on their relative magnitude, it is desirable to capture these readings in an ordered manner. Update maintains an array of flags, indicating which signals have been received during the current interval. It also maintains a list of readings using the following function:

```
(define update-readings
   (lambda (theta reading pop-count)
      (let ((new-cnt (- pop-count (length theta))))
         (append theta (make-list new-cnt reading)))))
```

Argument pop-count represents the number of signals received from distinct clocks during this interval. Since the argument for reading, $LC - Q$, is increasing with each tick of the local counter, this functions captures the readings of the remote clocks in a sorted list.

The DDD specification of the convergence function is:

```
(define cfn
   (lambda (theta)
      (floor_divby2 (add (index (f+1) theta)
                         (index (n-f) theta)))))

(define floor_divby2 (lambda (a) (floor (div a 2))))
```

This is a direct translation of the specification given in Section 2. Since we have been careful to capture the readings in sorted order, index simply selects the $(F + 1)$th and $(N - F)$th elements of the sorted list.

This specification of the synchronization algorithm is quite general. We could easily substitute a number of different convergence functions. For many of the published convergence functions, it would be unnecessary to alter our mechanism for capturing the readings of remote clocks in a sorted list. Any of the following convergence functions could be used instead of the fault-tolerant midpoint (the names are from Schneider [19]): Egocentric Average, Fast Convergence Algorithm, or Fault-tolerant Average. Other algorithms, such as that by Srikanth and Toueg [21], do not fit as cleanly into this specification.

Other portions of the algorithm depend upon our choice of convergence function. Since we are using the fault-tolerant midpoint, we know we will have enough readings to compute the convergence function as soon as we have seen signals from $N - F$ other clocks. A different convergence function would require a different predicate '`Enough(...)?`'. Similarly, simple inspection of the convergence function allows us to conclude that we can compute the convergence function within a single tick, so in our case predicate '`cfn computed?`' is always true.

During normal operation, the interval index i is incremented at the end of each interval. However, at system startup or when multiple faults occur, it may be appropriate to clear this index. Hence, we modify i using the abstract function `NEXT` in the ASM chart. For a system that enables recovery from transient faults, `NEXT` will also include a majority vote of indices received from other clocks. The elapsed time in clock ticks from the start of the protocol is $iR + LC$.

Now that we have described the architectural components of this behavioral specification, we can transform it into a structural description using DDD and begin refining the architecture. The initial architecture derived from this behavioral specification is given in Figure 3.

The areas enclosed by dotted lines in Figure 3 will remain unchanged throughout the remainder of this development. Function `test` generates signals to other clocks in the system. Correct implementation requires knowledge of delays in the underlying communication mechanism. Therefore, we will defer implementing this portion of the design until we build a complete system. Similarly, there are compelling reasons to defer implementing the population count module and the interval index counter. In particular, the function `NEXT` includes a vote of indices received from other clocks in the system. This function can either be realized in software or can be merged with the voter from a Byzantine agreement module.

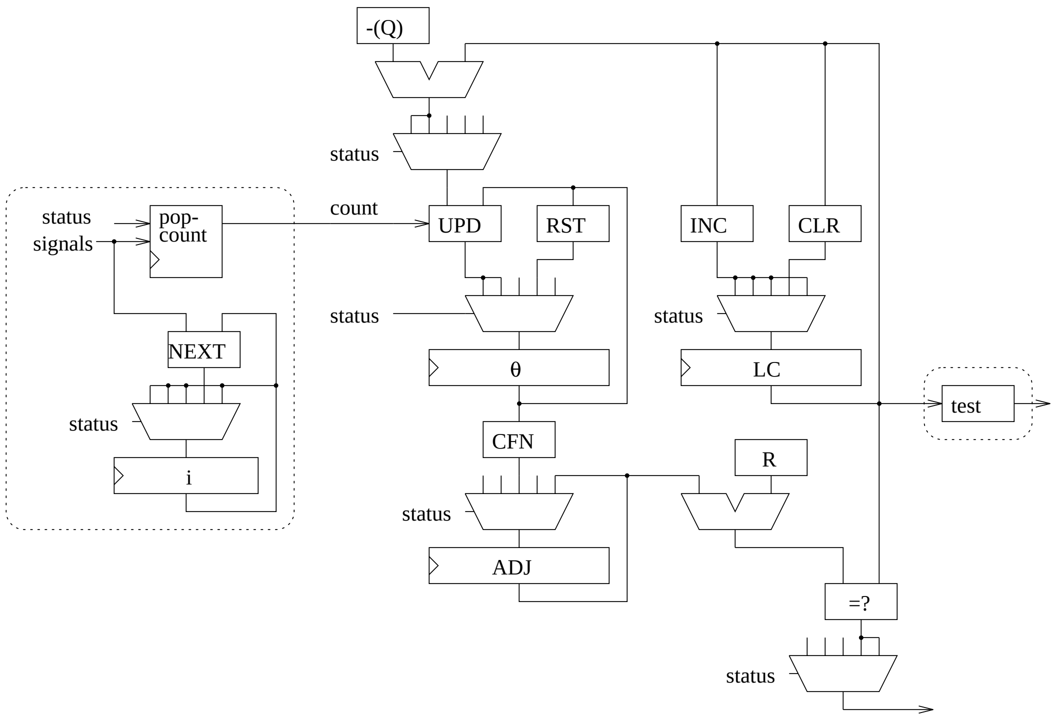

Figure 3: Initial Architecture

5 Structural Transformations

This section will focus on the core function of the synchronization system. Using standard DDD transformations augmented by some specialized techniques, we will develop a circuit design, central to the synchronization algorithm, that is independent of the number of participants in the protocol.

Since we know that the convergence function only requires two readings, it is not necessary to generate estimates for all clocks in the system. We would like to replace this portion of the design with a more efficient realization as depicted by the ad hoc transformation in Figure 4. This design step involves standard DDD transformations, such as expanding the definition of *cfn*, in addition to a step justified by mechanized proof. We first isolate the portion of the design which we want to transform. This is the part of Figure 4 enclosed by a dotted box. The DDD representation of the contents of this dotted box prior to the ad hoc transformation is given by:

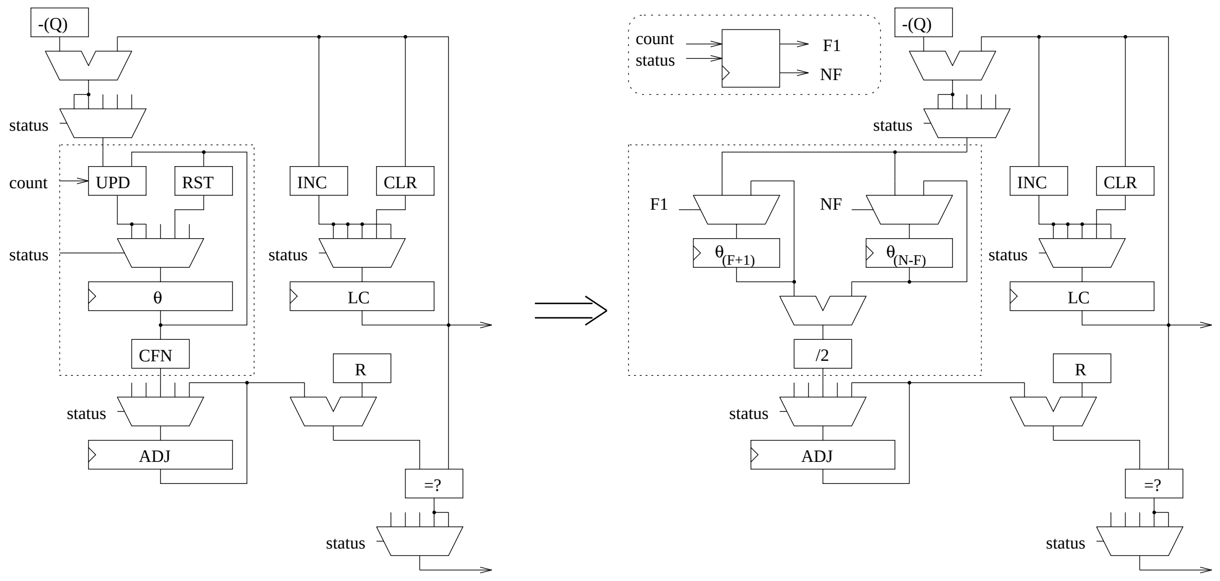

Figure 4: Ad Hoc Transformation

```
(define CLOCK-SYNC.1.2
   (lambda (ADJUST READING POP-COUNT END-OF-INTERVAL STATE)
      (system-letrec
         ((STATUS (XPS STATE ADJUST END-OF-INTERVAL))
          (THETA
               (! (new-theta)
                  (SELECT
                     STATUS
                     (UPDATE-READINGS THETA READING POP-COUNT)
                     (UPDATE-READINGS THETA READING POP-COUNT)
                     THETA
                     (RESET-THETA THETA)
                     THETA)))
          (CFN (SELECT
                  STATUS
                  ?
                  ?
                  (FLOOR_DIVBY2
                     (ADD (INDEX (F+1) THETA)
                           (INDEX (N-F) THETA)))
                  ?
                  ?)))
         (list STATUS THETA CFN))))
```

In the DDD descriptions, a stream equation (defining X) of the form $(X\ (!\ x\ (F\ X\ Y)))$ indicates that X is the output of a register. In more conventional formal notation, we would say that X is an infinite list satisfying the equation $X = \mathsf{cons}(x, F(X, Y))$. The stream THETA above is of this form. A stream without a "!", such as CFN above, represents the output of a combinational circuit element. We use the symbol '?' to represent a don't care value.

We wish to replace this sub-system with a more efficient hardware implementation. Our collection of clock readings, θ, is represented by a sorted list. We can replace this list, which would require N registers, with two registers, one for each value required in the computation of the convergence function.

Within DDD, we introduce two new streams, THETA-F1 and THETA-NF as follows:

```
( ...
   (F1 (!  ?  (GEQ POP-COUNT (F+1))))
   (NF (!  ?  (GEQ POP-COUNT (N-F))))
   (THETA-F1
      (!  ?  (UPDATE-REG F1 THETA-F1 READING)))
   (THETA-NF
      (!  ?  (UPDATE-REG NF THETA-NF READING)))
   ...  )
```

Both the initial sub-system description and these new streams are manually translated into the PVS logic. We also extract information characterizing the behavior of the input streams. The PVS proof shows that whenever the convergence function is computed, say at time j, that THETA-NF^j = (INDEX (N-F) THETA)^j (and similarly for THETA-F1). This proof depends critically upon the fact that POP-COUNT is non-decreasing over a synchronization interval and at least $N - F$ when the convergence function is computed.

The PVS proofs were independent of the values N and F. Any transformation purely within DDD would require a concrete specification of these two values. Furthermore, now that the design has been transformed to the two-register implementation, the streams `F1` and `NF` can be moved into a different partition of the design space. This leaves us with a core subsystem that is independent of the number of clocks in the system.

After performing the ad hoc transformation, we invoke some standard DDD transformations resulting in the final architecture depicted in Figure 5. The most interesting of these

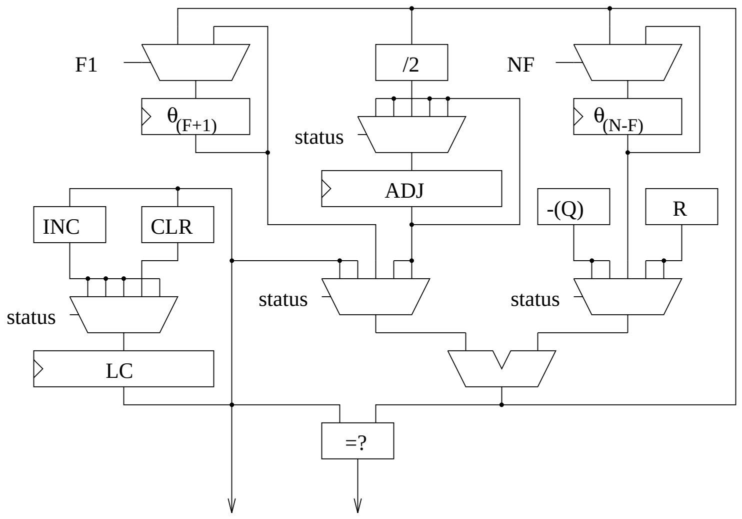

Figure 5: Final Architecture

transformations factors the three adders in the previous design into a single component. Now that we have a verified architecture, we can generate a concrete instance of the design.

6 Gate-level realization

The architecture in Figure 5 is still at an abstract level in that signals represent integer values. To obtain a concrete gate-level realization we need to instantiate these signals with bit-vectors of appropriate width. The values of Q and R are instantiated to 2^{12} and 2^{13} respectively. A 16-bit representation is sufficient to avoid overflow using these values.

The DDD-derived architecture can be mapped onto a bit-vector representation using the *projection* transformations of DDD. It can then be directly mapped onto Actel modules. We already have efficient Actel implementations of both an adder and an incrementor. In essence, we want to replace some modules of the derived architecture with efficient implementations of the same functions. The technique we use here is similar to the one reported in [4].

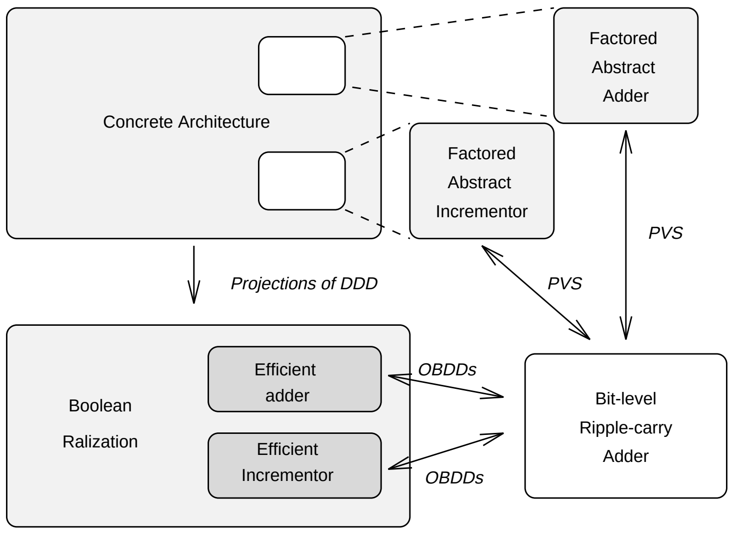

Figure 6: Boolean Realization

Since we would like to replace the adder and incrementor modules with our own designs, these modules are factored out prior to the *projection* transformations. The rest of the architecture is then projected onto a 16-bit representation, either by the standard projections of DDD or by other projections which are later proved to be correct with PVS. At this stage, the whole design, except for the factored adder and incrementor, is at the boolean level. To complete the transformation we have to show the equivalence of the factored modules and our engineered versions of the same functions.

The equivalence of the abstract adder and our efficient adder is established in two stages as shown in Figure 6. First we use PVS to prove that the abstract adder can be realized by a ripple-carry bit-vector adder. This is analogous to a *projection* transformation in DDD. Next, to complete the verification, the formal equivalence between the engineered gate-level version of the adder and the ripple-carry adder is established using OBDDs by encoding both the adders into boolean equations. The same kind of proof technique is used for the incrementor. This circuit can now be directly realized using an Actel FPGA.

7 Concluding Remarks

In this paper, we have presented the development of a fault-tolerant clock synchronization circuit using a combination of formal design tools. The primary tool was the Digital Design Derivation system, which provides a framework to explore design options in a safe manner. In the course of development, we recognized that there were design optimizations that could be more easily handled using either mechanical theorem proving techniques or tautology checkers. By incorporating these techniques, we developed a core circuit design that can be used in a clock synchronization system composed of an arbitrary number of redundant elements.

We were also able to defer implementation decisions for other aspects of the design. This flexibility is necessary in the development of a full architecture. The DDD system has proven effective in helping isolate portions of the design for further refinement. We believe that further extensions of this approach will allow us to integrate this synchronization circuit into a fault-tolerant architecture.

The verification approach spanned the spectrum from a high-level specification of abstract properties to a realization of a boolean circuit description. A mechanical theorem proving system is quite useful for the verification of abstract algorithms. DDD has proven effective in transforming an algorithmic function definition into a circuit description. Binary Decision Diagrams are quite useful in proving boolean equivalence. A practical formal design system should exploit the strengths of each of these approaches. Our presentation has illustrated some possible interactions between these different approaches. However, the interfaces between these formal systems still need to be formalized.

References

[1] William R. Bevier and William D. Young. The proof of correctness of a fault-tolerant circuit design. In *Second IFIP Conference on Dependable Computing For Critical Applications*, pages 107–114, Tucson, Arizona, February 1991.

[2] Bhaskar Bose. DDD - A Transformation system for Digital Design Derivation. Technical Report 331, Computer Science Dept. Indiana University, May 1991.

[3] 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.

[4] Bhaskar Bose, Steven D. Johnson, and Shyam Pullela. Integrating boolean verification with formal derivation. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of IFIP Conference on Hardware Description Languages and their Applications*, pages 127–134. Elsevier, April 1993.

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), September 1992.

[6] Ben L. Di Vito and Ricky W. Butler. Provable transient recovery for frame-based, fault-tolerant computing systems. In *Real-Time Systems Symposium*, Phoenix, Az, December 1992.

[7] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In *Dependable Computing for Critical Applications 2*, Dependable Computing and Fault-Tolerant Systems, pages 279–306. Springer Verlag, Wien New York, 1992.

[8] Warren A. Hunt. A formal HDL and its use in the FM9001 verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning in Hardware Design*. Prentice-Hall, 1992.

[9] Steven D. Johnson, B. Bose, and C.D. Boyer. A tactical framework for digital design. In Birtwistle and Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988.

[10] 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.

[11] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

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

[13] J Strother Moore. Mechanically verified hardware implementing an 8-bit parallel io byzantine agreement processor. NASA Contractor Report 189588, April 1992.

[14] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer Verlag.

[15] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design.* Prentice/Hall International, second edition, 1987.

[16] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In *Second International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 237–258. Springer Verlag, Nijmegen, The Netherlands, January 1992.

[17] 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.

[18] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. Research Report 78, DEC-SRC, September 1991.

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

[20] 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, Nijmegen, The Netherlands, January 1992.

[21] T.K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[22] Mandayam Srivas and Mark Bickford. Verification of the FtCayuga fault-tolerant microprocessor system: Volume 1: A case study in theorem prover-based verification. Contractor Report 4381, NASA, July 1991. Authors' affiliation: ORA Corporation, Ithaca, NY.

[23] Mandayam Srivas and Mark Bickford. Moving formal methods into practice: Verifying the FTPP scoreboard: Phase 1 results. NASA Contractor Report 189607, May 1992.

[24] M. Esen Tuna, Steven D. Johnson, and Bob Burger. Continuations in hardware-software codesign, March 1994. submitted to ICCD.

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