# A Light-Weight Framework for Hardware Verification

Christoph Kern, Tarik Ono-Tesfaye and Mark R. Greenstreet*

Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4
Canada
{ckern,tesfaye,mrg}@cs.ubc.ca

**Abstract.** *This paper describes a deductive verification framework that allows the use of general purpose decision procedures and traditional model checking along with domain specific inference rules. The latter allow established algorithms for timing verification and other hardware verification tasks to be imported into the verification framework. To demonstrate this approach, a SRT divider is verified using a transistor-level model with timing.*

## 1 Introduction

Most formal verification of hardware designs is based on state-space exploration or theorem proving. State space exploration provides an automatic approach for verifying properties of designs described by relatively small models. In principle, theorem proving techniques can be applied to much larger and more detailed design descriptions. However, the large demands for the time of expert users prevents the wide-scale application of theorem proving techniques.

The strengths and weaknesses of state-space exploration and theorem proving are in many ways complementary. This has motivated several recent efforts to combine the two techniques [4]. One approach is to embed state-space exploration algorithms as decision procedures in a general purpose theorem prover [20]. In this approach, the design and specification are represented by formulas in the logic of the prover, and decision procedures are oracles, introducing new theorems into the system. Alternatively, some researchers have augmented state-space exploration tools with simple theorem proving capability [11, 1, 18]. We extend this approach with the integration of *domain specific* decision procedures. For example, to verify timing properties of a divider, a simple, depth-first, tree-walk algorithm determines timing relations within the design. In principle, this algorithm could be formalized in a higher-order logic and proven correct; however, such an effort is not directly germane to the question of whether or not the divider is correct.

* This work was supported in part by NSERC research grant OGP-0138501, a NSERC graduate fellowship and a UBC graduate fellowship.

We view the verification task as one of maximizing the probability of producing a correct design subject to schedule and budget constraints. Using informal, domain-specific decision procedures in a deductive framework, we can verify critical properties of real designs that would not be practical to verify by theorem proving or model checking alone. Section 2 elaborates this claim. Section 3 describes our implementaion of this framework, and section 4 presents our verification of a self-timed divider using this tool.

### 1.1 Running Example: Asynchronous Divider Verification

Our divider verification establishes refinement between progressively more detailed descriptions of the design written in the Synchronized Transitions language [25]. The highest level model is an abstract specification of radix-2 SRT division on rational numbers; we prove functional correctness of the algorithm at this level. The most detailed model formalizes the transistor-level structure along with its timing properties. Each level of the hierarchy inherits the safety properties of the higher levels: by showing that the top-level model divides correctly, we establish that all of lower level the models divide correctly as well. Although there have been many published verifications of dividers, we believe that our work is distinguished by spanning the complete design hierarchy.

### 1.2 Synchronized Transitions

A Synchronized Transitions (abbr. ST) [24] program is an initial state predicate and a collection of transitions. A transition is a guarded command. For example,

$$\ll \texttt{x > y} \rightarrow \texttt{x, y := y, x} \gg$$

is a transition that is enabled to swap x and y when x is greater than y. Transitions may be combined using the asynchronous combinator, $\|$, for example $t_1 \| t_2 \| \ldots \| t_n$. Program execution consists of repeatedly selecting a transition, testing its guard, and, if the guard is satisfied, performing the multi-assignment. The order in which transitions are selected is unspecified: this non-determinism models arbitrary delays in a speed-independent model. ST provides other combinators and other language features which are not presented in this paper.

### 1.3 Semantics

We employ a *wp* semantics (see [7]) for ST. If $P$ is a program and $Q$ is a predicate, then $wp(P, Q)$ is the weakest condition that must hold such that $Q$ is guaranteed to hold after any single action allowed by $P$ is performed. Consider a transition $\ll G \rightarrow M \gg$: the guard, $G$, denotes a function from program states to the Booleans; the multi-assignment, $M$, denotes a function from states to states. A *wp* semantics of ST includes

$$wp(\ll G \rightarrow M \gg, Q) = G \Rightarrow Q \circ M$$
$$wp(t_1 \| t_2 \| \ldots \| t_n, Q) = \bigwedge_{i=1}^{n} wp(t_i, Q)$$

We make extensive use of **invariants**. A predicate $I$ is an invariant if $I$ holding in some state ensures that $I$ will hold in all possible subsequent states of the program. In particular, $I$ is an invariant of $P$ *iff* $I \Rightarrow wp(P, I)$. A predicate $Q$ is a **safety property** of $P$ if $Q$ holds in all states reachable in any execution of $P$. As shown in [12], $Q$ is a safety property of $P$ if and only if there is an invariant $I$ such that $Q_0 \Rightarrow I$ and $I \Rightarrow Q$.

Intuitively, program $P'$ is a **refinement** of $P$ if every reachable state transition that $P'$ can make corresponds to a move of $P$. More formally, refinement is defined with respect to an **abstraction mapping** $A$ that maps the states of $P'$ to $P$. $P'$ is a refinement of $P$ under abstraction mapping $A$ *iff* for every reachable state $s'_1$ of $P'$ and every state $s'_2$ that is reachable by performing a single transition of $P'$ from $s'_1$, either $A(s'_1) = A(s'_2)$ (a stuttering action), or there is a transiton of $P$ that effects a move from $A(s'_1)$ to $A(s'_2)$.

## 2 Verification Approach

Like many theorem provers, our verification tool presents a deductive style of verification. However, there are three ways in which our approach differs from traditional theorem proving:

**Integration of informal reasoning.** Domain-specific decision procedures can be used in our framework. Such procedures provide an algorithmic encapsulation of informal domain expertise.

**Syntactic embedding of the HDL.** Our framework favours an embedding of the hardware description language (e.g. ST) at a syntactic level. Inference rules operate directly on the HDL's abstract syntax.

**Merging of inference rules and decision procedures.** In traditional theorem provers, inference rules provide pattern-based rewriting of proof obligations, while decision procedures (if any) decide the validity of leaf obligations in a proof tree. In our framework, inference rules may perform non-trivial computations to decide the soundness of a proof step, or to derive the result of an inference step.

### 2.1 Informal reasoning in formal verification

At first, the suggestion of allowing informal reasoning to be introduced into a formal proof appears to be outrageous: if an informal inference rule is unsound, it can invalidate any proof in which the rule is used. However, informal rules provide a practical way to tailor our verification tool to specific domains and verify properties that would not be practical to address by strictly formal approaches. When errors are found in a design, the verification is worthwhile even if some steps are justified only informally.

Informal reasoning is commonplace in many verification efforts. For example, model-checking is typically applied to an abstraction of the design that was produced informally by a verification expert [10, 19]. Although the absence of errors in the abstraction does not guarantee the correctness of the actual design, errors found in the abstraction can reveal errors in the actual design. Many theorem-prover based verifications model functional units at the register transfer level: the gate- and transistor-levels of the design are validated only through simulation and informal reviews [23].

We make two uses of informal rules. First an informal rule can provide an algorithmic encoding of domain knowledge where a formalization in logic would be unacceptably time consuming. For example, the timing analysis procedure that we used derives a graph whose nodes correspond to the channel connected regions of the transistor-level circuit. The circuit topology is syntactically encoded in the text of the ST program, and the procedure derives timing bounds through graph traversal. The correspondence between the graph and the original circuit and the soundness of the graph traversal have only been shown informally.

Second, we use several 'semi-formal' rules for reasoning about ST programs. For instance, the proof rules for reasoning about invariants, safety properties, and refinements are founded on theorems that were formally proven (although the proofs have not been mechanically checked). These theorems are based on a formal semantics of a core language only, and their extension to the full language with records, arrays, functions, and modules is informal.

### 2.2 Syntactic embedding of the HDL

Verification requires a description of the design as a formula in the appropriate logic. Four ways to obtain such a description are:

1. *Describe the design directly in a general purpose logic.* With this approach, the full expressiveness of the logic is available to the designer [8]. However, many designers find such logics unfamiliar, and most tools for other aspects of the design process (e.g. simulation, synthesis, physical design, etc.) support special-purpose specification or hardware description languages.
2. *Deeply embed the HDL in the logic* [5]. With this approach, the syntax of the HDL as well as its semantic interpretation are defined within the logic. This provides a rigorous embedding where statements of the HDL become objects of an abstract data type in the logic, and proof-rules for the HDL can be proven as theorems in the logic, using the definitions of the semantic functions. The effort for producing such an embedding can be substantial, although it may be possible to amortize this effort over many designs.
3. *Shallowly embed the HDL in the logic.* With this approach, the translation from the HDL to the logic occurs outside of the logic. Shallow embeddings can be easier to implement than deep embeddings because the translation process is informal with a corresponding loss of rigour. Because program structures are not represented in the logic, theorems that refer to the syntactic structure of the HDL description can be neither stated nor proven [5].
4. *Syntactic embeddings.* In a syntactic embedding, the logic is extended to include proof-rules for the HDL, and the syntax of the HDL becomes part of the syntax of the logic. Because programs appear 'literally' in proof obligations, theorems about syntactic forms can be included as proof rules; results

of inferences can be interpreted in the original domain; and counter-examples can be presented in terms of the original program.

We use a syntactic embedding of ST into the logic of our theorem prover. The core logic of the prover is first-order with quantifiers, equality, and arithmetic operations. We extend this logic with inference and rewriting rules for the ST combinators, *wp*, invariants, safety properties, refinement, and timing-analysis.

We have found that a syntactic embedding simplifies the implementation of semi-formal or informal inference rules. Such rules are often based on syntactic analysis of the underlying program. These rules are easier to implement, and hopefully less prone to implementation errors, because the abstract syntax of the program is immediately available in the syntactic embedding. Furthermore, proof systems based on syntactic proof rules tend to match the designer's intuitive understanding of the HDL.

There are noteworthy drawbacks to this approach. Because the proof rules effectively become axioms of the logic, it is impossible to prove the consistency of the proof system. Doing so would require a formulation of a complete formal semantics of the logic, and thus the specification language, which is precisely what we are trying to avoid. Furthermore, as with shallow embeddings, no meta-reasoning about the semantics of the specification language is possible.

### 2.3 Merging of Decision Procedures and Inference Rules

Traditional mechanized theorem provers generally use only decision procedures in the classic sense of an algorithm that decides the validity of a formula. Such decision procedures are used to discharge proof obligations in a single automatic step, i.e. they operate on the leaves of a proof tree. Proof steps interior to the proof tree, however, are generally justified by matching them with an inference rule schema, and possibly checking side conditions or provisos.

We remove the restriction of decision procedures to leaf obligations and allow inference rules to use arbitrary algorithms to decide the soundness of a proof step. Theoretically, lifting this restriction has no significance; such an "inference procedure" can be replaced by the corresponding leaf decision procedures, and inferences using propositional logic. However, there are significant practical advantages to our approach. In many cases, it is convenient to let the inference rule compute the derived obligations rather than requiring the user to provide them. Of course, one could perform two computations of the derived obligation: one outside of the trusted core to derive the result for the user, and the other in the core to verify the result. Such an approach has obvious disadvantages with respect to efficiency and software maintenance. These problems would be particularly severe in a framework such as ours where it is supposed to be easy to add and extend domain-specific inference rules and decision procedures. Our "inference procedures" provide a simple mechanism for avoiding these problems.

## 3 Prototype Implementation

We have implemented a proof-of-concept proof checking environment for our approach. It has three architectural components. A generic core provides proof state and theorem objects, as well as a tactic interface. The second component is a library of common decision procedures, while the third comprises the code that is specific to a particular object logic. The system has been implemented in Standard ML of New Jersey [3]. As in HOL, ML also forms the user-interface for the proof checker.

### 3.1 Generic Core

Similar to theorem proving environments such as HOL, PVS or Isabelle [9, 16, 17], a (backwards-style) proof in our proof checker is represented by a sequence of proof states. A proof state consists of the claim, the pending obligations, and some bookkeeping information. The claim and obligations are judgements whcich can be, for instance, a sequent (in a sequent calculus), or a formula (in a natural deduction style calculus). In the initial proof state of a proof, the list of pending obligations consists only of the claim. Rules of inference are implemented as functions from proof state to proof state, and are used to transform one or more pending obligations into zero or more (simpler) obligations. The available proof rules are registered with the claim state and cannot be modified afterwards; in a sense, the become hypotheses of the theorem. This permits user-defined domain-specific proof rules to be introduced without modification of the core.

A proof state with no pending obligations corresponds to a proven claim, i.e. a **theorem**. To allow for theorems to be used in later proofs without having to check, and therefore execute, their proof before each use, we provide theorem objects, which associate a claim with a proof, i.e. a function that takes the claim proof state and returns a proof state with no pending obligations. Theorems can only be used in a proof if they were imported into the initial proof state. We provide facilities that analyze the dependency between theorems, ensure the absence of circularity, check all proofs that a theorem depends on, and generate reports.

All of the above components are parameterized in the syntax of the logic and a well-formedness predicate for proof obligations. The parameterization is realized through SML functors.

To facilitate the interactive development of proofs, we provide a simple goal package, which maintains a current proof-state to which rules can be applied, and allows proof steps to be undone. As indicated above, a proof in our system is a SML function from proof states to proof states. We provide a library of higher-order functions on proof rules (analogous to tacticals in e.g. HOL or Isabelle) which facilitate the construction of proofs from basic proof rules. This is often more effective and produces more readable proofs than writing a proof ‘from scratch’ in plain SML.

### 3.2 Library of Common Decision Procedures

This library comprises core routines of several commonly used decision procedures. The library is independent of a particular object logic; instantiating a decision procedure for a logic requires writing a small amount of interface code.

To support Boolean tautology checking as well as symbolic model checking, the library provides an abstract data type for boolean expressions in a canonical representation. The underlying implementation of this data type is a state-of-the art BDD package [22] that was integrated into the SML/NJ runtime system. The interface provides full access to the control aspects of the BDD package, such as variable reordering strategies, cache sizes etc. Based on the BDD package, we have implemented a package for symbolic manipulation of bit-vectors and arithmetic operation thereon.

Components for arithmetic decision procedures include a package for arbitrary precision integer and rational arithmetic, polynomials, and a decision procedure for linear arithmetic.

Based on these procedures, we have implemented a decision procedure that discharges arbitrary tautologies composed of linear inequalities with boolean connectives. We have not implemented a decision procedure for combinations of theories (e.g. [14, 21]) as our simple procedures were sufficient for the divider proof. All decision procedures include counter-example facilities for non-valid formulas.

### 3.3 Object Logic for Synchronized Transitions

We have instantiated the generic core with a logic suitable for reasoning about ST programs. The proof system is a sequent calculus for explicitly typed first-order logic that is extended with all types, constants and operators of ST, including transition-valued expressions.

As indicated in section 2.2, assertions on ST programs, such as invariants, safety properties and refinement, are formulated in terms of predicates on transition-valued expressions. We provide proof rules, such as the *wp*-based rule for invariants, that allow such obligations to be reduced to obligations that are purely within quantifier-free logic with boolean connectives, arithmetic, *If*-expressions, and arrays and records under store and select.

As an example, consider a proof state that includes the pending obligation:

$$\mathit{HasInvariant}(\ll i > 0 \rightarrow i := i-1 \gg \| \ll i < N \rightarrow i := i+1 \gg, 0 \leq i \leq N)$$

This obligation states that the two transitions maintain the given invariant. An application of the proof rule for *HasInvariant* rewrites this obligation as

$$(0 \leq i \leq N) \Rightarrow \mathit{wp}(\ll i > 0 \rightarrow i := i-1 \gg \| \ll i < N \rightarrow i := i+1 \gg, 0 \leq i \leq N)$$

An application of the proof rule for *wp* for $\|$ and two applications of the rule for *wp* for transitions yields:

$$
\begin{array}{l}
(0 \leq i \leq N) \Rightarrow ( \\
\quad\;\; ((i > 0) \Rightarrow (0 \leq i - 1 \leq N)) \\
\;\; \wedge ((i < N) \Rightarrow (0 \leq i + 1 \leq N)) \\
)
\end{array}
$$

This last obligation can be discharged using the decision procedure for linear inequalities with boolean connectives.

Further proof rules include the usual rules for sequent manipulations, rewrites, simplification and lifting of *If*-expressions, quantifier manipulations, and arithmetic simplifications. Together with decision procedures for propositional calculus and linear arithmetic, these are frequently sufficient to discharge obligations arising from assertions about ST programs. More specialized proof rules will be explained briefly in the context of the divider verification presented in the remainder of the paper.

## 4 Example: Proving a Self-Timed Divider Correct

We evaluated the proof checker by verifying Williams' self-timed divider [27], which implements the radix-2 SRT algorithm [6]. We reconstructed the design from the descriptions in [27] and [28]. A variation of this design is used incorporated in the HAL SPARC CPU.

### 4.1 Description of the Divider

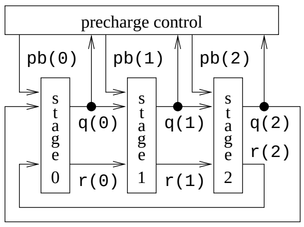


**Fig. 1.** Divider Architecture

As shown in figure 1, the divider consists of three identical stages, each of which performs the computation of a single iterative step of the SRT division algorithm, and which pass intermediate results around in a circular fashion. Each stage computes a new partial remainder (in carry-save representation) and quotient digit, based on the result of the previous iteration which it receives from the preceding stage. The design is self-timed, with signals encoded as dual-rail values [26], and implemented in precharged logic.

The precharge control block sequences the iterative computation. This block reads the stage completion signals and regulates the operation of the stages through the precharge control signals. Taken together, the precharge control block and the dual-rail data paths form a ring oscillator; three steps of the SRT algorithm are computed in each period of the oscillation.

According to the precharge control signals, each stage can be in one of three states: precharge, evaluate, or hold. The "precharge bar" signal for stage `i` is `pb(i)`. When `pb(i)` is low, stage `i` is precharging. Precharging leads to a state where every dual-rail signal produced by the stage has the "empty" value. Evaluation leads to a state where every signal has a "valid" value. A stage in the holding state leaves its outputs unchanged so that its successor can use them to compute the next partial remainder and quotient bit. A simple invariant that captures this sequencing is central in many of our proofs.

Williams employed two optimizations to improve the performance of the divider. First, he assumed that a stage can precharge faster than its predecessor can evaluate. Second, he assumed that the quotient bit of a stage will be the last output to change during the evaluation stage. The first optimization allows stage `i+1` to precharge in parallel with the evaluation phase of stage `i`. If no timing assumptions were made, these operations would have to be performed sequentially. The second optimization allows the computation of stage `i+1` to start as soon as the quotient bit from stage `i` is output, without any extra hardware to check the completion status of the partial remainder. Due to these optimizations, verifying the functionality of the divider includes proof obligations that require timing analysis. This timing analysis establishes relative orderings of events in the operation of the divider and shows that the assumptions on which the optimizations are based are indeed correct.

### 4.2 A Refinement Hierarchy for the Divider

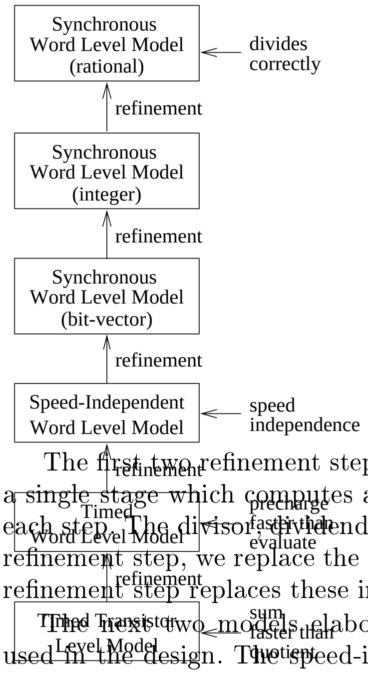


**Fig. 2.** Verification Hierarchy

The transistor-level model of the divider is too large to permit model checking, and too complicated to verify from first principles using a theorem prover. Therefore it is desirable to prove safety properties on a more abstract, higher-level model and show that these properties hold in the more detailed models. We used a hierarchy of models as depicted in figure 2 to verify the divider. Arrows indicate verification obligations: vertical arrows correspond to refinement proofs, horizontal arrows indicate other properties that either establish correctness or assist in the refinement proofs.

The first two refinement steps are data-refinements. Our top-level model has a single stage which computes a quotient bit and the next partial remainder in each step. The divisor, dividend, and remainder have rational values. In the first refinement step, we replace the rational values with integer values, and the next refinement step replaces these integers with bit-vectors.

The next two models elaborate upon hte self-time handshaking protocols used in the design. The speed-independent model has three divider stages and implements a handshaking protocol that does not depend on the timing delays of the components. In the timed, word-level model, bounds are given on the ratio of precharge time to evaluation time.

The lowest-level model corresponds directly to our transistor-level implementation of the divider chip. Variables in this model are represented using dual-rail code. In the higher level models, the remainder word was computed as a single, atomic action. Here, each bit is set independently. In this transistor-level model, a stage's completion status is determined solely by the quotient bit output.

### 4.3 Functional Correctness

In radix-2 SRT division, each quotient bit can have the value -1, 0 or 1 (see [6]). If the current remainder $R_i$ is greater or equal to 0, 1 is a valid quotient digit choice. If the remainder is negative, -1 is a valid choice for the next quotient digit. If $2|R_i| \leq$ divisor, the quotient digit can also be 0. In our synchronous model of the divider this overlapping choice for the digit is represented by three transitions combined with the asynchronous combinator:

$$\begin{array}{lcll} & \ll & 0 \leq R_i & \rightarrow q_i := 1 & \gg \\ \| & \ll & -D \leq 2 * R_i \leq D & \rightarrow q_i := 0 & \gg \\ \| & \ll & R_i < 0 & \rightarrow q_i := -1 & \gg \end{array}$$

For example, if the current remainder is equal to $-0.2 * D$, then either the first or the second transition may be chosen for the next step. By using a non-determinism, we avoid cluttering this description with implementation details. Deterministic quotient bit selection is introduced in the synchronous, bit-vector model.

The following two properties are invariants of the synchronous divider model:

$$\begin{array}{lcl} (i) & |R_i| & \leq & D \\ (ii) & 2C - D\sum_{j=0}^{i-1} q_j 2^{-j} & = & R_i 2^{1-i}, \end{array}$$

where $R_i$ is the remainder determined in iteration $i$. From these two invariants and the initial condition that the divisor $D$ and dividend $C$ are normalized to lie in the ranges $\frac{1}{2} \leq D < 1$ and $0 < C < D$, we easily proved that the computed quotient $\sum_{j=0}^{i-1} q_j 2^{-j}$ asymptotically approaches the true quotient $C/D$.

### 4.4 Refinement Proofs

This section gives short overviews of the refinement proofs and mentions key problems within each proof. It is this chain of refinement proofs which establishes that the functional correctness proven on the abstract, synchronous model also applies to the transistor-level model. The divider models will be referred to as rational divider, integer divider, bit-vector divider, speed-independent divider, timed divider and transistor-level divider.

In our approach, refinement is a safety-property. To establish refinement, we must first show that initial states of the lower-level model correspond to legal, initial states of the higher-level model. Then, we must show that for each transition that can be performed by the lower-level model, there is a corresponding transition of the higher-level model, or that it is a stuttering move [13] of the higher-level model. These proof obligations are derived automatically by one of the proof rules that encodes the semantics of our logic for ST.

Because refinement is a safety property, we can assume that if the state of the lower-level model before a transition is performed maps to a state of the higher-level model, it satisfies any safety properties that have been established for higher-level model. This allows us to use safety properties of the higher-level model in the proof of refinement. This property is very helpful for our proofs. For example, arithmetic properties that are established for the top-level models can be used when verifying the other models. Likewise, invariants that are established by model-checking on intermediate level models can be used when verifying lower-level models. Because of this, the verification of refinement is often simply a matter of tautology checking.

**Refinement between the Rational Divider and the Integer Divider.** To convert the integer values in the integer divider to the rational valued variables in the rational divider one has to simply apply a division by $2^{N-1}$. To prove that the integer-valued divider is a refinement of the rational-valued one, it needs to be shown that overflows do not happen. However, this is implied by the safety property $|R_i| * \frac{1}{2^{N-1}} \leq D * \frac{1}{2^{N-1}}$ which the integer divider model inherits from the rational divider.

**Refinement between the Integer Divider and the Bit-Vector Divider.** In the bit-vector divider, carry-save representation is used for the remainder value. The abstraction mapping adds the carry and sum words to determine the remainder value at the integer level. Furthermore, the next quotient digit is computed deterministically in the bit-vector model based on the top bits of the carry-save adder without resolving the carry of the bottom bits. For the refinement proof it needs to be shown that for each quotient digit choice of the bit-vector model, an equivalent choice can be made by the higher-level model.

Several safety properties of the higher-level models are used to bound the values of the divider and partial remainder at each iteration. Combined with properties of the abstraction mapping, refinement is straightforward to show. The proof obligations were discharged by the combination of a proof rule that reduces arithmetic operations on bit-vectors to BDDs, and the BDD-based tautology checker.

When we first attempted this proof, we discovered quotient selections that could lead to a violation on clause of the word-level invariant that bounds the magnitude of the partial remainder. Using the counter-examples generated by our decision procedures, we identified the cause, and modified the quotient selection to correct the error. As previously indicated, we reconstructed our models published descriptions [27, 28]. It is likely that this error is an artifact of our interpretation of these sources. We hope to have a clarification of this issue for the final paper.

**Refinement between the Bit-Vector Divider and Speed-Independent Divider.** The speed-independent model consists of three divider stages and all control is performed by explicit handshaking without any timing assumptions.

For the abstraction mapping it is necessary to determine which stage's output of the synchronous model's only stage. Intuitively, the precharge control ensures that at any time, there is a stage whose output value is the last partial remainder computed, and this stage can be identified by the state of the precharge control. We verified a hand-written invariant to show that the control logic operates as intended. We then wrote an abstraction function that selected the appropriate output value for the partial remainder based on the state of the precharge control. Using this abstraction function, the refinement property was easily proven.

**Refinement between the Speed-Independent Divider and the Timed Divider.** In the speed-independent model, the precharge control block performs an explicit check to ensure that stage `i+1` is done precharging (i.e. its outputs are empty) before stage `i` starts evaluating. The timed model starts both operation s in parallel, and timing bounds are used to show that precharging completes before evaluation. This corresponds to William's first optimization in the design of the chip, as discussed in section 4.1.

We use the approach of [2] to model time: a real-valued variable is added to the program to model the current time, transistion guards are strengthened to express lower bounds on delays, and an action for advancing time is defined so as to observe upper bounds on delays (i.e. time may not progress beyond the maximum delay for a pending action). In this model, the clause of the guard for the evaluate action that asserted that the successor stage is replaced by a clause that states that the successor stage started precharging sufficiently far in the past. We then verified an invariant that implies that whenever this timing condition is satisfied, the successor stage has finished precharging. With this invariant, refinement was easily verified (see [15] for details).

**Refinement between the Timed Divider and Transistor-Level Divider.** To establish that the transistor-level model implements the timed divider, two major problems have to be addressed. First, the dual-rail encoded signals of the transistor-level model must be mapped to the bit-vectors of the timed divider. Second, in the transistor-level model only the quotient bit output is used to determine if a stage has finished evaluation. It therefore needs to be shown that the quotient bit of a stage becomes valid only after all other outputs of a stage are valid. This corresponds to William's second optimization as mentioned in section 4.1.

The first problem was addressed by defining an appropriate abstraction mapping. Solving the second problem requires an argument about the timing of events as data values propagate from a stages inputs through its logic elements after it enters evaluation mode. Our verification adapted a simple depth-first graph traversal algorithm for timing verification of combinational logic for use in the self-timed context. Thus, our timing verification procedure verifies that the timing graph has the desired property and generates side obligations stating e.g. that the inputs to a stage remain stable while it is in evaluation mode. We verified these conditions by introducing a side-hierarchy that matched the handshaking of the original hierarchy with the details of the computation abstracted away. The required properties were established by model-checking on the side-hierarchy and showing refinement between the transistor-level models of the side-hierarchy and the main hierarchy. See [15] for details on the timing analysis and this refinement proof.

## 5 Conclusions

We have demonstrated an approach to the verification of hardware designs that combines deductive reasoning with algorithmic decision procedures. Like theorem provers such as HOL, Isabelle or PVS, our tool employs the notion of proof states, to which a sequence of inference rules and decision procedures is applied to form a proof. The most important distinction between our tool and more traditional provers is that the set of available inference rules and decision procedures is not fixed, but may be extended with domain-specific rules. This permits reasoning that would be unacceptably costly to formalize rigorously in logic to be introduced into a correctness argument in a controlled manner.

We have demonstrated the practical applicability of our approach by carrying out a top-to-bottom verification of a non-trivial hardware design, a self-timed implementation of SRT division. Our verification connects a high-level specification of the SRT division algorithm with a formalization of the transistor-level implementation through a series of refinement proofs. Safety-properties proven at the highest level, in particular correct division, are propagated down the chain of refinements and thus hold for the implementation. The proof obligations arising from the safety property and refinement proofs varied widely in nature, from arithmetic obligations at the algorithmic level to timing properties at the transistor level. Although there have been many published verifications of dividers, we believe that our work is distinguished by spanning the complete design hierarchy. Domain-specific proof rules such as the timing-verification procedure played a crucial role in achieving this.

**Acknowledgements**

We would like to thank Alan Hu for many helpful discussions on the divider verification. Our thanks to Ted Williams who explained many details of his design to one of the authors several years ago. And thanks to Andrew Appel and Lorenz Huelsbergen for answering our questions on integrating the CUDD package into SML.

## References

1. Mark Aagaard and Carl-Johan H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *Int. Conf. on Computer-Aided Design, ICCAD '95*, pages 7–10, November 1995.
2. Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. In J.W. de Bakker et al., editors, *Proceedings of the REX Workshop, "Real-Time: Theory in Practice"*. Springer, 1992. LNCS 600.

3. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *3rd Int. Symp. on Prog. Lang. Implement. and Logic Program.*, number 528 in Lect. Notes Comput. Sci., pages 1–13. Springer-Verlag, August 1991.
4. N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *8th Int. Conf. Computer-Aided Verification, CAV '96*, number 1102 in Lect. Notes Comput. Sci., pages 415–418. Springer-Verlag, July/August 1996.
5. Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *1st Int. Conf. on Theorem Provers in Circuit Design, TPCD '92*, pages 129–156. North Holland, June 1992.
6. Joseph J.F. Cavanagh. *Digital computer arithmetic : design and implementation*. McGraw-Hill, New York, 1984.
7. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
8. Michael Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Science Publishers, 1985.
9. Michael J.C. Gordon. HOL: a proof generating system for higher-order logic. In Graham Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 74–128. Kluwer Academic Publishers, 1988.
10. Cheryl Harkness and Elizabeth Wolf. Verifying the Summit bus converter protocols with symbolic model checking. *Formal Meth. System Design*, 4:83–97, 1994.
11. Scott Hazelhurst and Carl-Johan H. Seger. A simple theorem prover based on symbolic trajectory evaluation and BDDs. *IEEE Trans. Comput. Aided Des. Integr. Circuits*, 14(4):413–422, April 1995.
12. Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. Technical Report 17, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, May 1987.
13. Leslie Lamport. The temporal logic of actions. Technical Report 71, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, December 1991.
14. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.
15. Tarik Ono-Tesfaye, Christoph Kern, and Mark R. Greenstreet. Verifying a self-timed divider. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1998.
16. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th Int. Conf. Automated Deduction (CADE '92)*, number 607 in Lect. Notes Comput. Sci., pages 748–752. Springer-Verlag, 1992.
17. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lect. Notes Comput. Sci. Springer-Verlag, Berlin, 1994.
18. Amir Pnueli and Elad Shahar. A platform for combining deductive with algorithmic verificication. In Rajeev Alur and Thomas A. Henzinger, editors, *8th Int. Conf. Computer-Aided Verification, CAV '96*, number 1102 in Lect. Notes Comput. Sci., pages 184–195. Springer-Verlag, July/August 1996.
19. F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In S. Haridi, K. Ali, and P. Magnusson, editors, *Proc. EURO-Par '95 Parallel Processing*, number 966 in Lect. Notes Comput. Sci., pages 287–300. Springer-Verlag, August 1995.

20. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *7th Int. Conf. Computer-Aided Verification, CAV '95*, number 939 in Lect. Notes Comput. Sci., pages 84–97. Springer-Verlag, July 1995.
21. Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, January 1984.
22. Fabio Somenzi. CUDD: CU Decision Diagram Package. URL: `http://bessie.colorado.edu/~fabio/CUDD/ cuddIntro.html`.
23. Mandayam K. Srivas and Steven P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Meth. System Design*, 8(2):153–188, March 1996.
24. Jørgen Staunstrup. *A Formal Approach to Hardware Design*. Kluwer, 1993.
25. Jørgen Staunstrup. *A formal approach to hardware design*. Kluwer Academic Publishers, Boston, 1994.
26. Tom Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, 3:1–8, 1988.
27. T. E. Williams, M. A. Horowitz, R. L. Alverson, and T. S . Yang. A self-timed chip for division. In *Stanford Conference on Advanced Research in VLSI*, pages 75–96, March 1987.
28. Ted E. Williams. Self-timed rings and their application to division. Technical Report CSL-TR-91-482, Computer Systems Lab, Dept. of EE, Stanford, May 1991.