Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Light-Weight Framework for Hardware Verification

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

Dept. of Computer Science, University of British Columbia
Vancouver, BC V6T 1Z4, Canada
e-mail: {ckern,tesfaye,mrg}@cs.ubc.ca

Received: date / Revised version: date

Abstract. We have developed a verification framework that combines deductive reasoning, general purpose decision procedures, and domain-specific reasoning. This paper describes this framework and presents a case study in which we verified a SRT divider circuit. Our proof starts with a high-level description of the SRT algorithm on rational numbers. We verified the correctness of the algorithm. With a sequence of five refinement proofs, we established that a transistor-level implementation with timing is a refinement of a high-level specification of the high-level division algorithm.

Our approach is made practical by integrating formal theorem proving techniques with informal domain-specific reasoning. User-defined inference rules provide domain specific decision procedures, while an LCF-style, first-order-logic theorem prover allows results from these procedures to be combined into a complete proof. Including these "semi-formal" rules as hypotheses of the theorems in which they are used preserves the logical validity of the proofs and tracks and documents the use of domain-specific reasoning.

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 prevent 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. One approach is to embed state-space exploration algorithms as decision procedures in a general purpose theorem prover [41, 21, 22]. 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 [24, 1, 6, 39]. We follow the first approach.

Designers can usually provide explanations for why they expect their designs to work; we seek to exploit this domain expertise in our verification. These explanations are typically "informal" in the sense that they are expressed in language or notation that lacks a complete and precise semantics. Furthermore, these arguments typically focus on key principles while ignoring technical side conditions. These technical details are essential for the construction of a rigorous proof, but they often render formal methods unusable by most designers. For example, if one operation always completes before another, the designer should be able to clearly state this property. However, we do not expect the designer to have the time or the mathematical expertise to be able to express standard timing verification algorithms as formal mathematical arguments.

Domain specific decision procedures can discharge many of the tedious obligations of a formal proof. These procedures may be "informal" in the sense that they may not perform an explicit sequence of logical transformations to reduce proof obligations to basic axioms or previously proven theorems. Furthermore, the decision procedures or their implementations may not come with a proof of correctness. Examples of such decision procedures include binary decision diagrams (BDDs) for deciding predicate logic formulas, syntactic transformation rules for reasoning about programs, and graph-based timing verification algorithms. Although each of these could in principle be formalized in terms of basic mathematical axioms, the effort to do so is not justified for many, if not most, designs. For example, although the algorithms

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

for manipulating BDDs have been proven correct [11], we are not aware of any *implementation* of these algorithms that has a correctness proof. Most BDD packages including the one that we used [46] are highly optimized for performance and therefore not particularly amenable for formal verification. On the other hand, BDDs are probably among the most formal of the informal procedures that we consider. For example, we use a graph-based timing analysis algorithm whose justification is informal. Precisely mapping the operations of the algorithm into the semantics of our object logic would require a great deal of effort and divert our attention from examining more probable sources of errors. Rather than rejecting informal procedures, we provide a controlled mechanism for including informal procedures as explicit hypotheses to the theorems we prove.

Many properties that designers want to verify require a combination of timing verification, equivalence checking, model checking, and other techniques, some general, and some domain specific. Accordingly, we provide a simple theorem prover for first order logic that allows the results from such point tools to be combined in a systematic fashion to verify important system properties. Our intent is that the deductive arguments required to combine these results should closely mimic the informal arguments of the designer. When such arguments fail, we should be able to quickly identify the error in the reasoning or the design.

We view the verification task as one of maximizing the probability of producing a correct design subject to schedule and budget constraints. Using *domain-specific* and possibly *informal* decision procedures and inference rules in a deductive framework, we can verify critical properties of real designs that would not be practical to verify by theorem proving and/or model checking alone. Section 2 elaborates this claim, and section 3 describes our implementation of this framework. Section 4 describes the self-timed chip that we verified as a case study for our approach. Sections 5 and 6 present our verification of this divider.

1.1 Running Example: Asynchronous Divider Verification

We have implemented a proof-of-concept tool based on the methodology presented in this paper and used it to verify the design of a self-timed divider [50]. Our divider verification establishes refinement between progressively more detailed descriptions of the design written in the Synchronized Transitions language [48]. 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 the lower level models divide correctly as well. The highest level model is an abstract specification of radix-2 SRT [14] division on rational numbers. This algorithm is similar to the binary version of the traditional "paper-and-pencil" algorithm except that quotient bits are chosen from the set $\{-1, 0, 1\}$ instead of the more traditional $\{0, 1\}$. As described in section 4.2, such redundancy is used in nearly all hardware dividers because it facilitates efficient implementations. We prove functional correctness of the algorithm at this level. The most detailed model includes the transistor-level structure along with its timing properties. The transistor-level structure is extracted syntactically from the Synchronized Transitions program. By inheriting safety properties from the higher levels, the timing verification problem becomes relatively straightforward, although far too tedious to perform manually. We found that a simple, conservative, graph-traversal algorithm was sufficient to verify the necessary timing properties for the divider. Section 6.6.2 describes our integration of timing verification into our tool.

Because each lower levels of the refinement hierarchy inherit safety properties from the higher levels, the transistor level model inherits the "correct division" property from the top-level description. Thus, our verification establishes that the timed, transistor-level model divides correctly.

1.2 Synchronized Transitions

Synchronized Transitions (abbr. ST) [48] is a programming language based on the paradigm of guarded actions executing on a state space [18]. A ST program consists of

– a collection of variables whose domains define the state space associated with the program,
– a predicate on states defining the initial set of states,
– and a collection of transitions.

A transition in turn consists of a guard (a predicate on the state space which determines if the transition is enabled in a given state) and a multi-assignment which determines state reached by executing this transition. For example,

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

is a transition that is enabled to exchange the values of the variables x and y when x is greater than y. A transition written without a guard is always enabled; a transition written without a multi-assignment leaves the state space unchanged when executed.

Transitions may be combined using the asynchronous combinator, $\|$; the strong synchronous combinator, $*$, or the weak synchronous combinator, $+$. For example, the ST program $(t_1 \| t_2)$ consists of two transitions, t_1 and t_2, composed with the asynchronous combinator. 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. If two transitions are composed with the strong synchronous combinator, the multi-assignments can be performed as a single atomic action if both guards are satisfied. If either guard is not satisfied, then no action is enabled. Operationally, performing two or more transitions as a single atomic action means that all reads for the guards and multi-assignments are performed before any writes. Thus,

$$\begin{aligned} &\ll g_1 \rightarrow l_1 \texttt{:=} r_1 \gg * \ll g_2 \rightarrow l_2 \texttt{:=} r_2 \gg \\ \equiv &\ll g_1 \wedge g_2 \rightarrow l_1, l_2 \texttt{:=} r_1, r_2 \gg \end{aligned}$$

If two transitions are composed with the weak synchronous combinator, then if one of the transitions is enabled, the corresponding multi-assignment can be performed, and if both transitions are enabled, then both multi-assignments can be performed as a single atomic action. In other words, the multi-assignments all enabled transitions composed with the weak synchronous combinator are performed as a single, atomic action. Formally,

$$\begin{array}{ll} & \ll g_1 \to l_1 := r_1 \gg + \ll g_2 \to l_2 := r_2 \gg \\ \equiv & \ll \neg g_1 \wedge \neg g_2 \gg \\ & \| \ll g_1 \wedge \neg g_2 \to l_1 := r_1 \gg \\ & \| \ll \neg g_1 \wedge g_2 \to l_2 := r_2 \gg \\ & \| \ll g_1 \wedge g_2 \to l_1, l_2 := r_1, r_2 \gg \end{array}$$

1.3 Semantics

We employ a *wp* semantics (see [18]) 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. Let $\mathcal{S}$ be the state-space of a program P. Consider a transition $\ll G \to M \gg$ of P: the guard, G, denotes a predicate $\underline{G} : \mathcal{S} \to \{true, false\}$. The multi-assignment, M, denotes a function $\underline{M} : \mathcal{S} \to \mathcal{S}$. A *wp* semantics of ST includes

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

where $\circ$ denotes function composition.

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 [27], 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 (see [2]). 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 transition of P that effects a move from $A(s'_1)$ to $A(s'_2)$.

1.4 Related Work

The highly publicized Pentium division bug [7] has been followed by active research in the verification of floating point calculations in general and division in particular. Clarke *et. al.* [16] verified an implementation of the radix-4 SRT algorithm. The radix-4 algorithm computes two bits of the quotient per iteration whereas the simpler, radix-2 algorithm used in the divider that we consider computes one bit per iteration. Like our approach, Clarke *et. al.* made liberal use of outside decision procedures. In particular, their theorem prover, Analytica [17], is written in the Mathematica command language [52] and uses Mathematica's symbolic computation facilities. Although Mathematica is based on well-defined mathematical concepts, the implementation has not been subject to any form of formal verification. Clarke *et. al.* modeled the hardware at a level of fairly large functional units such as adders and look-up tables.

Rueß *et. al.* have described a verification of an implementation of the radix-4 SRT algorithm [42]. They used the PVS [36] theorem prover for their verification. Like Clarke *et. al.*, they modeled the hardware at a level of large functional units. The proof consisted of an initial case split followed by evaluation of ground predicates. This strategy exploits the tightly integrated decision procedures of PVS.

Moore *et. al.* [32] studied a different implementation of division, namely the microcode implementation of the AMD K5 microprocessor using the Nqthm/ACL-2 theorem prover [26]. The K5 uses Newton-Raphson iteration instead of the SRT algorithm. The distinctive achievement of the Moore *et. al.* proof is that they showed compliance with the the IEEE floating point standard [25] including its various rounding mode and support for denormalized numbers. Moore has conjectured [31] that the facilities of a large theorem prover such as Nqthm/ACL-2, HOL98, or PVS are necessary to manage the intricacies of the floating point standard.

Aagaard and Seger verified a floating point multiplier using gate-level models [1]. They used trajectory evaluation for their verification, and the properties that they verified were stated as trajectory formulas. These formulas corresponded directly to the properties in the IEEE specification. Although we believe that the ensemble of properties that they proved establishes implementation of the IEEE standard, the limitations of trajectory formulas prevented them from stating a simple, top-level theorem that "obviously" captures the designer's intent.

The verification presented in this paper is unique because of the mechanisms that our framework provides to integrate informal, heuristic decision procedures into a simple, theorem proving environment. This approach allowed us to verify a divider starting with a simple theorem of functional correctness and eventually showing that this correctness is preserved by a model at the level of transistors modeled as switches. In particular, we were able to verify several timing assumptions made by the designer in order to optimize the performance of the divider.

2 Verification Approach

Like many theorem provers, our verification tool presents a deductive style of verification. Our approach differs from traditional theorem proving in three crucial ways:

Integration of informal reasoning. Our framework supports the inclusion of domain-specific decision procedures and inference rules. Such procedures provide an algorithmic encapsulation of formal or informal domain expertise; this allows domain expertise to be introduced as hypotheses of a proof. The framework keeps track of the set of inference rules used in a proof.

Syntactic embedding of the HDL. Our framework favors an embedding of the hardware description language (HDL) 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 effort is worthwhile even if some steps are justified only informally.

A potentially insidious danger of using informal inference rules is that a faulty decision procedure could render contradictory judgements. The user could then derive the theorem "false" which can be used to prove any other assertion. Here, the simplicity of our theorem prover offers some protection: if a user finds such a contradiction and uses it to prove an unrelated theorem, then he or she deserves the unpleasant fate that awaits. We suspect that the dangers of "informal" decision procedures would be more serious in a more powerful deductive framework such as Nqthm/ACL2 [10] or the tightly integrated decision procedures of PVS [35] where complicated chains of inference can be performed without specific direction from the user. We are not arguing against the benefits of more powerful theorem provers; we simply note that the integration of new decision procedures can be more difficult when working with sophisticated theorem proving heuristics (see [9]).

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 [23,40]. 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-level models of the design are validated only through simulation and informal reviews [47].

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, we used a timing analysis procedure that 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.

In our framework, informal inference rules and decision procedures can be seen as a generalization of the concept of using a hypothesis in a proof: Usually, a hypothesis is simply a formula that is assumed to be valid. An informal rule in contrast is an algorithm; the corresponding hypothesis is that it only generates sound inferences.

Since the soundness of a proof depends on the soundness of the inference rules used in its construction, one's confidence in the truth of a theorem verified in our framework will depend on one's confidence in the soundness and correct implementation of the inference rules used. Our framework recognizes this and includes a mechanism which allows a user to track the set of rules used in the proof of a particular theorem. When a theorem is posed, the set of inference rules which can be used in its proof must be stated; any attempt to use any other rule will be rejected by the system. The system can be queried to recursively compute the set of inference rules used in the proof of a particular theorem, and other theorems used in its proof.

2.2 *Syntactic Embedding of the HDL*

Formal verification requires a description of the design as a formula in the appropriate logic. If it is not practical to describe the design directly in logic [19], e.g. because of lack of tool support for simulation, synthesis etc, an embedding of the HDL in the logic has to be devised. Such embeddings are commonly divided into two classes: deep and shallow [8]. In a *deep embedding*, both the (abstract) syntax of the HDL as well as its semantic interpretation are defined within the logic in terms of an abstract data type and a semantic function, respectively. This provides a very rigorous embedding and allows meta-reasoning about the HDL semantics. However, the effort for producing such an embedding can be substantial, although it may be possible to amortize this effort over many designs.

In a *shallow embedding* in contrast, the semantic interpretation of the HDL occurs outside the logic. Shallow embeddings can be easier to implement than deep embeddings because the translation process is informal with a corresponding loss of rigor. 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 [8].

We propose a third variant, a *syntactic embedding*: The syntax of the HDL becomes part of the syntax of the logic (see section 3.4 for the embedding of ST). As in a shallow embedding, the semantic interpretation is informal. However, the procedures that perform this interpretation are encapsulated as domain-specific inference rules. This provides a tighter integration with the prover than could be achieved with a shallow embedding. In particular, decision procedures can consider the syntactic structure of the program being verified. However, as with shallow embeddings, no meta-reasoning about the semantics of the specification language is possible.

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.

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.

Consider using a proof rule which implements the *wp* semantics for ST. Suppose we want to prove that a predicate is an invariant of a program and arrive at the obligation

$$\begin{array}{l} n \leq 10 \\ \Rightarrow wp(\ll n < 10 \rightarrow n := n+1 \gg, n \leq 10) \end{array}$$

The proof rule can rewrite this obligation to the equivalent obligation

$$\begin{array}{l} n \leq 10 \\ \Rightarrow (n < 10 \Rightarrow n+1 \leq 10) \end{array}$$

by computing the expression for $wp(P, Q)$ based on the semantic rules for *wp*. Note that computing the expression for $wp(P, Q)$ is much more straightforward than verifying that an arbitrary expression (supplied by the user) is equivalent to $wp(P, Q)$.

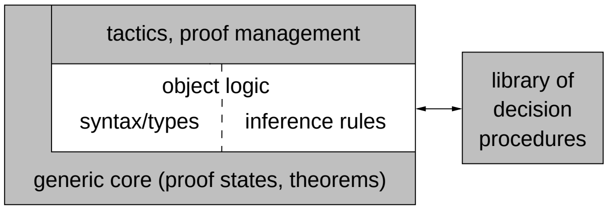

tactics, proof management
object logic
syntax/types
inference rules
generic core (proof states, theorems)
library of decision procedures

Fig. 1. Proof Checker Architecture

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 ease of adding and extending domain-specific inference rules and decision procedures is important. Our "inference procedures" provide a simple mechanism for avoiding these problems.

3 Prototype Implementation

We have implemented a proof-of-concept verification environment for our approach. Figure 1 shows the main architectural components of our tool. The system's core provides infrastructure for managing proof states and theorem objects. The implementation of this core is generic in nature, i.e., independent of a particular object logic. Generic components are depicted as shaded boxes in figure 1.

The object-logic-specific components of the checker include data-structures representing the abstract syntax and types of the logic, and the proof rules of the logic. The latter make use of a library of commonly used decision procedures.

A goal and proof tactic package provide an interface between the user and the core as well as the object logic. Proofs are represented as functions in the implementation language of our tool, Standard ML (SML) of New Jersey [4], which also acts as the user interface for proof development.

While an existing generic theorem proving environment (such Isabelle [37]) may have been a viable option for use as the core of our system, we have chosen to design and implement our own infrastructure. This decision was based on the observation that implementing both our approach of syntactically embedding the object logic as well as an interface for domain-specific decision procedures into an existing environment is a non-trivial exercise; we felt that making use of a core that is designed with our specific needs in mind was the preferable option.

3.1 Generic Core

Similar to theorem proving environments such as HOL, PVS or Isabelle [20,36,37], a (backwards-style) proof in our proof

```
signature JUDGEMENT = sig
    (* abstract type for judgements *)
    type form
    (* the well-formedness predicate *)
    val wellFormed : form -> bool

    (* abstract type for the environment in which a judgement
       is to be interpreted (e.g. symbol tables) *)
    type environment
    val setEnvironment : environment -> unit
    (* ... *)
end

signature PROOF_STATE = sig
    structure Judgement : JUDGEMENT
    (* a judgement with an identifier attached *)
    type  namedForm = string * Judgement.form

    (* abstract types for proof states and theorems *)
    type proofState
    type theorem

    (* Proof rule functions *)
    type proofRuleFn = {obls    : namedForm list,   newObls      : Judgement.form list,
                        auxInfo : exn,              importedThms : theorem list,
                        bookKp  : Judgement.bookkeepingInfo}
                    -> {newObls : namedForm list,   bookKp  : Judgement.bookkeepingInfo}
    (* a proof rule is a proofruleFn with a name *)
    type proofRule = {name:string, rule:proofRuleFn}

    (* create a goal state from a claim.  Only proof rules and theorems
       imported here can be used in the proof *)
    val goalState : {claim     : namedForm,     env      : Judgement.environment,
                     theorems  : theorem list,  rules    : proofRule list}
                  -> proofState

    (* datatype used to select obligations by index or identifier *)
    datatype obligationSpec = Idx of int | Nm of string
    (* apply the proof rule named to the obligations selected *)
    val applyProofRule : (string * proofState * obligationSpec list *
                          Judgement.form list * exn) -> proofState

    (* a proof is a function from proof states to proof states *)
    type proof = proofState -> proofState
    (* stating a theorem associates a proof with a claim *)
    val stateTheorem : {name : string, proof : proof, pstate : proofState }
                       -> theorem
    (* ... *)
end
```

Fig. 2. SML signatures for generic core

checker is represented as a sequence of proof states. A proof state consists of the claim, the pending obligations, and some bookkeeping information. The claim and obligations are judgments which can be, for instance, a sequent (in a sequent calculus), or a formula (in a natural deduction style calculus). A proof begins with an initial proof state in which 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, they 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. A theorem object associates 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. Figure 2 shows the SML signatures corresponding to the representation of judgements, proof states and theorems (for the sake of presentation, some details have been omitted). SML signatures act as interface specifications for SML structures (the SML notion of a module).

Signature `JUDGEMENT` provides an abstract interface between the domain-specific aspects of the proof infrastructure and its generic core. This abstract view consists of a type `form` for judgements, the predicate `wellFormed`, and a type `environment`. The environment mechanism allows the imperative-style implementation of entities, such as global symbol tables, that are not conveniently expressed in a functional style, as attributes of an object of type `form`. The core maintains this environment and guarantees that `setEnvironment` is invoked before any operation on a `form` is performed.

The generic implementation of proof states and theorems is realized as a SML functor that returns a structure that matches signature `PROOF_STATE` and takes a structure that matches `JUDGEMENT` as argument:

```
functor FProofState (structure F : JUDGEMENT)
        : PROOF_STATE =
struct (*....*) end
```

The SML type system ensures that only types specified in signature `PROOF_STATE` are exported. The type `proofState` is exported as an abstract type, which ensures that the only way to create an object of this type is using one of the functions in signature `PROOF_STATE` that returns such an object. Function `goalState` is the only function that creates a fresh `proofState`. No function alters the list of imported `theorems` and `proofRules` of a `proofState`. Function `applyProofRule` is the only one that modifies the list of pending obligations; it does so by looking up (in the proof state) the `proofRuleFn` indicated by the first argument, applying it to the specified obligations, and replacing these obligations with the returned new ones. Additional arguments are available for rule-specific auxiliary information (such as instantiations with a quantifier rule) and persistent bookkeeping information.

As indicated above, proofs in our system are SML functions 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 (which correspond to HOL tactics). Figure 3 shows a listing of the SML signature of the tactics interface. Type `tactic` is the type of functions from proof state to proof state. Function `elseTAC`, for example, is a higher order function that, given a pair of tactics, returns a new tactic. The tactic returned is a function that applies the first tactic to its proof state argument, and if this fails, returns the result of applying the second tactic. Failure of applying a tactic is communicated through SML exceptions. Similarly, `repeatTAC` repeatedly applies a tactic to a proofstate, until this tactic fails. Tacticals for other commonly used compositions of proof steps are provided as well.

As is apparent from figure 3, `tactic`s apply to entire proof states, i.e. the tactic may act on all pending obligations. However, many proof rules apply to a single obligation only, specified by a `PS.obligationSpec`. Such a proof step is represented as a `tactickle`, and a corresponding set of higher order functions for compositions of `tactickle`s is available.

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.

The implementations of both the tactic and goal package are generic, and realized as SML functors that take a structure matching signature `PROOF_STATE` as an argument.

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 Binary Decision Diagram (BDD) [12] package [46] 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 operations 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 an implementation of the simplex linear programming algorithm on arbitrary precision rational numbers.

Based on these library components, we have implemented a decision procedure that discharges arbitrary tautologies composed of linear inequalities with boolean connectives. The decision procedure is based on rewriting the negation of a formula into sums-of-products and refuting each product term [44]. If the original formula was a Boolean combination of linear inequalities, each of the products thus obtained defines the constraints of a linear program; refuting a product term is then equivalent to showing that the linear program has no solution. So far, we have not felt the need to implement a decision procedure for combinations of theories and/or un-

```
signature TACTIC = sig
    structure PS : PROOF_STATE

    (* a tactic is a proof step *)
    type tactic = PS.proofState -> PS.proofState
    (* elseTAC is the tactic that applies the first tactic, and if
       this fails the second one *)
    val elseTAC : (tactic * tactic) -> tactic
    (* repeatTAC is the tactic that repeatedly applies the given
       tactic until it fails *)
    val repeatTAC : tactic -> tactic
    (* ... *)

    (* tactics apply to entire proof states, tactickles to single
       obligations, but are otherwise analogous *)
    type tactickle = PS.obligationSpec -> tactic
    val elseTac : tactickle
    val repeatTac : (tactickle * tactickle) -> tactickle
    (* ... *)
end
```

Fig. 3. SML signature for tactic interface

interpreted functions (e.g. [33,45]) as our simple procedures were sufficient for the divider proof.

All decision procedures include counterexample facilities for non-valid formulas. For example, if a boolean formula composed of linear inequalities is not a tautology, then the decision procedure will exhibit a valuation of the variables for which the formula is not satisfied. With the decision procedure we are using for linear arithmetic, a counterexample is obtained as a byproduct of the proof attempt: Recall that the decision procedure proves a formula by rewriting its negation into a disjunction of conjunctions of linear constraints and showing that each such set of constraints is unsatisfiable. Unsatisfiability is shown by attempting to find a solution to a linear program with a corresponding set of constraints. If a solution is indeed found, this solution is a counterexample for the original formula.

Likewise, if the model checker rejects a safety property, it will provide a trace that starts in a state satisfying the initial state predicate, and ends in a state that violates the safety property. The model checker is based on reachable state set computation [13]; counterexamples are constructed by picking states from the sets of states defined by the iterations of the reachable state set computation. Each transition of the trace is labeled with the transition of the model that effects it. Counterexamples such as these allow the human verifier to quickly identify errors in the design or in the intended proof so that the appropriate corrections can be made.

3.3 Embedding of BDDs in Object Logic

Our BDD library provides facilities that allow the syntactic embedding of an object logic to be defined such that Boolan subexpressions can be represented by a BDD that is embedded into the expression itself. We have implemented an interface layer on top of the basic BDDs that provides a BDD datatype whose variables are (Boolean-typed) formulas of the object logic. Furthermore, such a BDD can appear as a (Boolean-type) subexpression of a formula of our logic. Thus, a BDD can appear anywhere in place of a Boolean formula, and vice versa. Note that many Boolean formulas do not have a compact BDD representation. For example, the formulas that relate a dividend and divisor to their quotient and remainder have sizes that are exponential in the number of bits of their operands. Having a choice of representation allows us to exploit the efficiency of BDDs where applicable; subexpressions that do have a compact BDD can be represented as such, while the top-level structure of the expression remains explicitly represented.

This mixed representation is useful for example in conjunction with a proof rule that rewrites Boolean-valued subexpressions involving bit-vectors into an equivalent predicate on individual bits by expanding bit-vector operations into their corresponding bitwise logical/arithmetic operations. In general, later proof steps would reason about this predicate using a BDD-based decision procedure for propositional logic, which requires the predicate to be transformed into an equivalent BDD.

Instead of constructing a (potentially large) formula for the predicate which is later rewritten into a BDD, in our framework the proof rule can directly and efficiently construct the BDD, and replace the original subformula of the obligation with this BDD.

3.4 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.

The abstract syntax of the logic is represented in the usual manner [38] as a SML datatype with constructors corresponding to literals, identifiers and operators. Type-checking of formulas is carried out by a function that recursively traverses a formula and annotates each well-typed formula with its type, and raises an exception for ill-typed subformulas. Once computed, the type of each subformula is cached for efficiency. Since the logic is explicitly typed (i.e. all identifiers need to be declared with their type), type-checking is efficiently decidable.

The well-formedness predicate for this logic states that all formulas that of a sequent must be well-typed and of Boolean type.

For the prototype implementation, we have avoided the effort of writing a parser for the language, and instead use SML's parser by declaring, at the SML top-level, a constructor with the appropriate fixity for each ST operator. ST source modules are transliterated into this form by a translator that shares its front-end with a compiler that compiles ST into executable code for simulation. An additional benefit of this choice is that the well-tested existing front-end provides independent syntax and type checking of the ST programs.

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. The latter is achieved by rules which rewrite operations on individual cells of an array (or components of a record) into operations on the entire array (record) as a single variable. For instance, an assignment

$$A[5] := b$$

would be written as

$$A := store(A, 5, b),$$

Where $store(A, 5, b)$ denotes the array A with the cell at index 5 updated with value b.

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

$$\begin{aligned} HasInvariant(\quad & \ll i > 0 \rightarrow i \mathtt{:=} i-1 \gg \\ & \| \ll i < N \rightarrow i \mathtt{:=} i+1 \gg, \\ & 0 \le i \le N) \end{aligned}$$

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

$$\begin{aligned} & (0 \le i \le N) \Rightarrow \\ & \quad wp(\quad \ll i > 0 \rightarrow i \mathtt{:=} i-1 \gg \\ & \quad\quad \| \ll i < N \rightarrow i \mathtt{:=} i+1 \gg, \\ & \quad 0 \le i \le N) \end{aligned}$$

An application of the proof rule for *wp*, which implements the semantics given in section 1.3, yields:

$$\begin{aligned} & (0 \le i \le N) \Rightarrow \\ & \quad (\quad ((i > 0) \Rightarrow (0 \le i-1 \le N)) \\ & \quad \wedge ((i < N) \Rightarrow (0 \le i+1 \le N))) \end{aligned}$$

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

As explained in the previous section, proof rules are invoked by supplying the name of the proof rule to the function `applyProofRule` of signature `PROOF_STATE`. For each proof rule, we provide a wrapper function returning a `tactic` or `tactickle`, given auxiliary arguments if applicable. For example, proof rule `ByTheorem`, which inserts a theorem identified by its name into the list of antecedents of a sequent, is encapsulated as a wrapper function with signature

```
val byTheorem : string -> tactickle
```

These wrapper functions are necessary for interaction with the tactics package. For instance, the proof that carries out the proof steps described above can be succinctly written as

```
(        invariantByWp
 thenTac linArith)
```

Here, `invariantByWp` is a compound tactickle that applies the first two steps, while `linArith` is a wrapper function that directly invokes the proof rule that implements our decision procedure for linear arithmetic. Given two tactickles A and B, $(A \mathtt{thenTac} B)$ is the tactickle that applies first A and then B, and fails if either of the two steps fails.

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 The Self-Timed Divider: Overview

We evaluated the proof checker by verifying Williams' self-timed divider [50], which implements the radix-2 SRT algorithm [14]. We reconstructed the design from the descriptions in [50] and [51]. A variation of this design is incorporated in the HAL SPARC CPU.

The verification of the divider while implementing the proof-of-concept tool in parallel took two people about four months. Finding appropriate invariants turned out to be surprisingly easy, usually only requiring a small number of iterations.

This section presents an overview of the divider design. Sections 4.1, 4.2, 4.3 are tutorial in nature describing carry-save addition, SRT division, and precharged logic respectively. Section 4.4 describes the particular divider from [50].

4.1 Carry-Save Arithmetic

Figure 4 shows three common ways to implement two's complement addition (see [14]). In all three cases, the adders receive as inputs two words, `a` and `b` of `n` bits each, Each adder

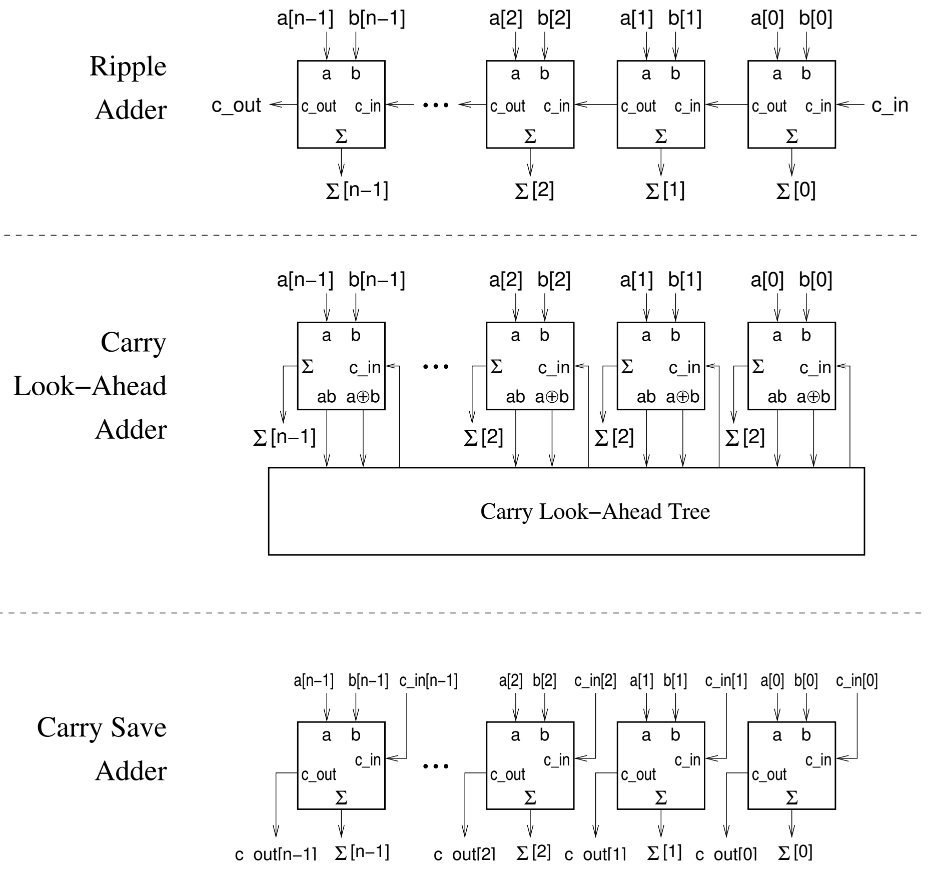


Fig. 4. Three Common Adder Designs

outputs an n-bit sum word, Σ. Each adder consists of n one-bit adders, where the i^{th} one-bit adder computes the sum of input bits `a[`i`]` and `b[`i`]` and outputs bit Σ`[`i`]`. The differences between the three designs lay in the ways that carries are handled.

The simplest adder is a carry-ripple. Each one-bit adder receives its carry-input from the previous stage (or the environment for adder for the least significant bit). This implementation resembles the traditional "paper-and-pencil" method: before computing the sum and carry at bit i, the carry from bit $i-1$ is required. In the worst-case, $O(n)$ time is required to compute an `n`-bit sum. As described in section 4.2, the SRT division algorithm requires $O(n)$ additions Thus, SRT division using carry-ripple adders requires $O(n^2)$ time for worst-case data value.

Carry look-ahead adders can compute an n-bit sum in $O(n \log n)$ time. While there are many variations on the design, all use some kind of tree structure to determine the carry-input values for each stage. While carry look-ahead approaches are very useful for general-purpose arithmetic units, carry-save adders provide higher performance from a simpler, smaller circuit for application where a large number consecutive of additions or subtractions are required. Division is just such an application.

In a carry-save adder, each one-bit adder receives its carry input from the adder for the previous bit of the *previous* adder.

decimal	non−redundant binary	carry−save
12	00001100	00 00 00 00 10 10 00 00
+ 13	+ 00001101	+ 0 0 0 0 1 1 0 1
25	00011001	00 00 00 01 01 00 00 10
+ 14	+ 00001110	+ 0 0 0 0 1 1 1 0
39	00100111	00 00 00 11 00 10 10 10
+ 15	+ 00001111	+ 0 0 0 00 1 1 1 1
54	00110110	00 00 01 00 11 01 01 00
+ 16	+ 00010000	+ 0 0 0 1 0 0 0 0
70	01000110	00 00 01 11 00 10 10 00

Fig. 5. Carry-Save Addition

Basically, the carry-save design takes advantage of the fact that addition is associative and commutative. Therefore, the carries can be added to the partial sums in any order to produce the same final result. Figure 5 shows ripple carry and carry-save addition when computing $12+13+14+15+16$. The outputs of the carry-save adders are represented with two bits per binary digit. The left bit in a pair for digit i represents $\Sigma[i]$; the right bit represents `c_out[i-1]`. The advantage of carry-save addition is that each addition can be computed in $O(1)$ time. As the SRT division algorithm requires $O(n)$ additions, SRT division using carry-save adders requires $O(n)$ time.

The complication introduced by carry-save addition is that it is a redundant representation: each binary digit of a value

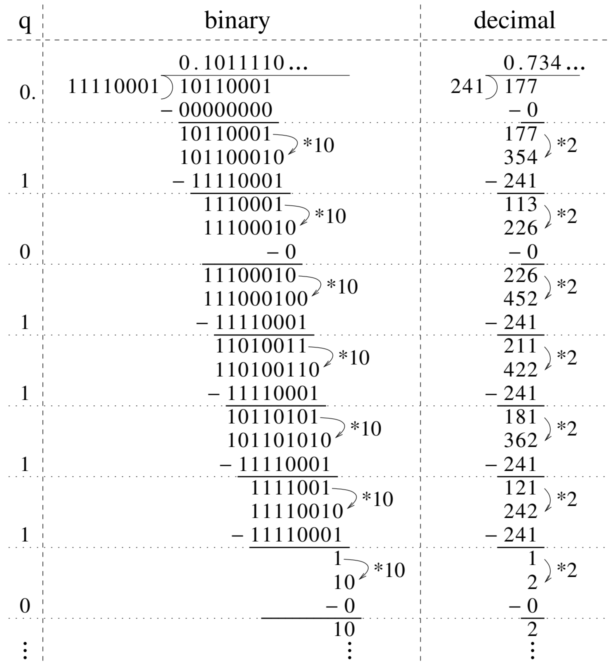


Fig. 6. Radix-2 Long Division

is represented by two bits. Accordingly, the same number can be represented with several different encodings. For example, every four-bit value has sixteen representations; the sixteen four-bit representations of the integer 5 are shown below:

00 01 00 01	00 01 00 10	00 00 11 01	00 00 11 10
00 10 00 01	00 10 00 10	01 11 11 01	01 11 11 10
11 01 00 01	11 01 00 10	11 00 11 01	11 00 11 10
11 10 00 01	11 10 00 10	10 11 11 01	10 11 11 10

The traditional, non-redundant, value can be extracted by forming two words, one for the partial sum, and one for the carries. The sixteen encodings of 5 shown above correspond to:

$0 +_{16} 5$	$1 +_{16} 4$	$2 +_{16} 3$	$3 +_{16} 2$
$4 +_{16} 1$	$5 +_{16} 0$	$6 +_{16} 15$	$7 +_{16} 14$
$8 +_{16} 13$	$9 +_{16} 12$	$10 +_{16} 11$	$11 +_{16} 10$
$12 +_{16} 9$	$13 +_{16} 8$	$14 +_{16} 7$	$15 +_{16} 6$

where $+_{16}$ denotes addition modulo 16. Likewise, a non-redundant result for the example from figure 5, is obtained by computing 00110110 + 00010000 which yields 01000110, the value computed by the ripple adder. For SRT division, this final add can be computed by either a ripple-carry or a carry-lookahead adder and preserve the $O(n)$ time requirement.

4.2 The SRT Division Algorithm

To motivate the SRT algorithm, first consider the traditional, "long-division" algorithm with binary arguments. Figure 6 shows binary, long division for computing $177 \div 241$. The left column (labeled "q") shows the quotient bits as they are calculated, the column labeled "binary" shows the calculations in binary notation, and the column labeled "decimal" shows the same calculations in decimal notation. The algorithm starts with a comparison of 177 and 241. Because 177 is less than 241, the integer part of the quotient is zero. Subtracting $0 * 241$ from 177 yields 177, which is the first partial remainder for the calculation. Let d denote the divisor, r denote the current partial remainder, and b denote the radix of the algorithm (for binary calculations, the $b = 2$). The computation of the quotient proceeds by repeatedly applying the following three steps:

1. Multiply the partial remainder by the radix: $p = b * r$.
2. Select the next quotient bit: $q = \lfloor p/d \rfloor$.
3. Compute the next partial remainder: $r = p - q * d$.

For radix-2 division, step 1 can be implemented by shifting the bits of r one position to the left; step 2 is a comparison of p and d (i.e. $q = 0$ if $p < d$ and $q = 1$ if $p \geq d$); step 3 is a subtraction. When carry-save arithmetic is used, the comparison in step 2 of the long division algorithm becomes difficult. Performing an exact comparison basically requires propagating all "pending carries" in the carry-save word to produce a non-redundant value, which would nullify the advantages of the carry-save approach.

The SRT algorithm implements division efficiently using carry-save arithmetic. SRT employs a redundant representation of the quotient – for the radix-2 algorithm, a quotient digit can be -1, 0, or 1. Each iteration of the SRT algorithm is similar to an iteration of the long division computation: the partial remainder is multiplied by the radix and compared with the divisor; a quotient digit is selected; and the product of the quotient digit and the divisor is subtracted from the scaled partial remainder to produce a new remainder. The redundancy in the quotient digits allows the comparison of the divisor and the partial remainder to be approximate, which in turn allows carry-save arithmetic to be used. As long as the magnitude of the partial remainder remains less than or equal to d, there is a sequence of quotient digits that drives the partial remainder to zero. With the radix-2 SRT algorith, one quotient bit is computed per iteration; accordingly, if the quotient is requied to the same precision as the operands, SRT division performs $O(n)$ additions to compute a quotient given n bit operands.

A charming feature of the radix-2 SRT algorithm is that valid quotient digits can be selected by only considering the four most-significant binary digits of the partial remainder, even when the partial remainder is represented in the carry-save style. We will describe the particular implementation that we verified. Our verification shows the correctness of the mantissa calculation of the divider. The operands to the divider are normalized floating point numbers[1]. Therefore, we assume that the divisor and dividend mantissas are in the interval $[0.5, 1)$. A critical invariant of the divider (verified in section 6.1) is $|r| \leq d < 1$. Figure 7 shows the valid choices of the quotient digit such that this invariant is maintained.

The divider uses a two's complement, carry-save representation of the partial remainders. The most significant bit in

[1] IEEE floating point [25] allows for "denormalized numbers." Typical implementations of this standard include additional mechanisms for handling denormalized divisors with a loss of performance. In this paper, we assume that the divisor is normalized.

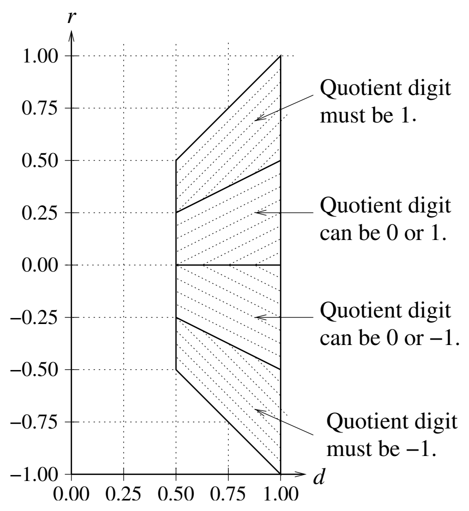


Fig. 7. Allowable quotient digit values for radix-2 SRT division

CRA sum	r	q
0000	$[0.00, +0.50)$	$+1$
0001	$[+0.25, +0.75)$	$+1$
0010	$[+0.50, +1.00)$	$+1$
0011	$[+0.75, +1.00)$	$+1$
1011	$(-1.00, -0.75)$	-1
1100	$(-1.00, -0.50)$	-1
1101	$[-0.75, -0.25)$	-1
1110	$[-0.50, 0.00)$	-1
1111	$[-0.25, +0.25)$	0

Table 1. Quotient bit selection for the radix-2 SRT divider

our implementation has weight -2. If a non-redundant representation were used for the partial remainder, then this bit would be unnecessary. With carry-save addition, this bit helps disambiguate the value of the partial remainder as we explain below.

Assume that the partial remainder, r, has n digits with $n \geq 4$. The rational number value of r can be estimated by using a four-bit, carry-ripple adder to compute the non-redundant, two's complement value for the top four digits of r. This ignores the remaining $n-4$ digits. Because two's complement notation is used, this neglected term must be positive or zero. The most-significant digit of the partial remainder has weight -2. Therefore, the fourth most significant digit has weight $1/4$. The neglected term is the sum of two words, each of which has a value of less than $1/4$. Therefore, the neglected term has a value in the interval $[0, 0.5)$.

This redundancy allows quotient bits to be selected without first propagating all carries in the partial remainder calculation. Instead, the top few bits of the partial remainder are calculated, deferring carry propagation from lower bits until subsequent iteration. Based on the top few bits of the partial remainder, a small range that contains all possible values of the partial remainder can be determined. The redundant representation allows a quotient digit to be selected that is valid for any value of the partial remainder in this range. After the complete quotient has been computed, the traditional, non-redundant representation can be obtained by performing a single subtraction operation. For floating point applications, the mantissa of the divisor can be assumed to be normalized; therefore it lies in the interval $[0.5, 1)$. For the radix-2 algorithm, quotient digits can be selected based on the value of the partial remainder without considering the value of the divisor.

Table 1 shows the quotient selection used in the divider presented in this paper. The column labeled "CRA sum" gives the output of the four bit carry-ripple adder described above. The column labeled r gives bounds on the value value of the partial remainder given the value of CRA sum, and the column labeled q gives the quotient digit for this value of CRA sum. For example, if the CRA sum is 0000, then the partial remainder must be in the interval $[0, 0.5)$. As shown in figure 7, a quotient digit of $+1$ is allowed whenever the partial remainder is non-negative, justifying this entry in table 1. If the CRA sum is 1011, then the 2's complement value of the CRA sum is -1.25. In this case, the invariant $|r| < 1$ establishes the lower bound for r. This means that there *must* be pending carries in the neglected part of the partial remainder word; in other words, the neglected part has a value of at least $1/4$. If the CRA sum is 1111, the sign of the partial remainder depends on the the pending carries from the neglected part of the partial remainder word. In this case, we note that the magnitude of the partial remainder is at most $1/4$. From figure 7, a quotient digit of 0 is allowed for any such partial remainder. This is the value chosen in the table. The explanations for the other entries in table 1 are similar to those stated above. There are no entries for sums from 0100 through 1011; the invariant $|r| < 1$ precludes the occurence of such values.

Figure 8 shows the calculations of the SRT algorithm when computing $\frac{177/256}{241/256}$ – this corresponds to the long division example from figure 6 with the divisor and dividend scaled to be in $[0, 0.5)$. The calculations employ 10-bit, two's complement to represent the divisor, the dividend, and the partial remainders. The (redundant) quotient is

(0).(+1)(-1)(+1)(0)(0)(0)(-1)(0)(+1)(-1)(-1)(-1).

To obtain the traditional (non-redundant) quotient, we subtract the -1's word from the 1's word, i.e.:

$$\begin{array}{r} 0.101000001000_2 \\ -0.010000100111_2 \\ \hline 0.010111100001_2 \\ = 0.73486328125_{10} \end{array}$$

4.3 Precharged Logic

Our verification starts with a proof of correctness for the radix-2 SRT algorithm described above and then progresses through a sequence of five refinement proofs to produce a verification of at timed transistor-level model. To prepare for these detailed models, we now describe the general style of logic circuits used in the divider.

We start with a one paragraph review of switch models for transistors (see, e.g. [49] for a more complete exposition). Figure 9 shows an n-channel MOSFET transistor

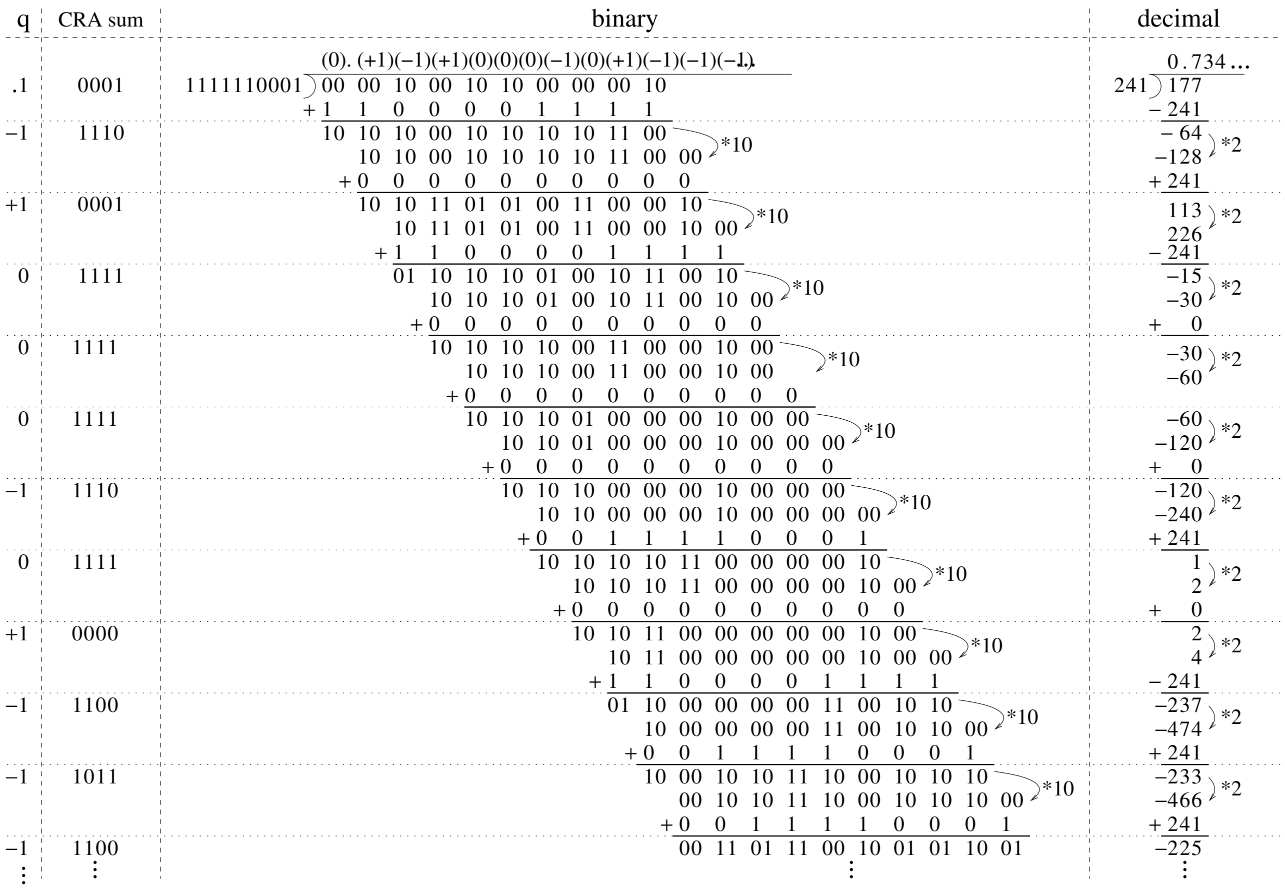


Fig. 8. SRT Division Example

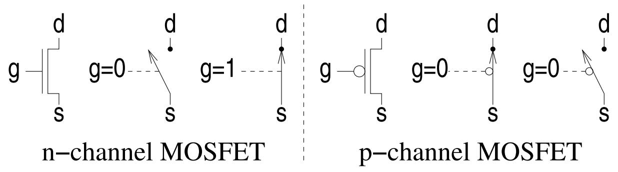


Fig. 9. Switch models for transistors

and a p-channel MOSFET transistor along with their switch level models. Each transistor has three terminals: the gate, the source, and the drain, labeled g, s, and d respectively in the figure. The source and drain are interchangeable; two different names exist for historical reasons. In a simple model, the transistor operates as a switch controlled by the voltage on the gate. For an n-channel transistor, a connection is made between the source and the drain when the gate is at a high voltage, and no connection is made when the gate is at a low voltage P-channel devices operate with a reversed sense of the drain: when the drain is high, no connection is made; when the drain is low, a connection is made.

The divider that we verified makes extensive use of *precharged* logic. Figure 10 shows a generic precharged logic gate and a specific example, an AND-OR gate. Precharged gates operate using a cycle of three phases. The first phase is the *precharge* phase. In this phase, the logic inputs (i.e. a, b, c, . . .) are low, and the "precharge bar" (i.e. negated precharge) signal, pb, is asserted by setting pb low. This creates a connection between node x and the power supply (labeled 1 in the figure). The pull-up transistor is made large enough that it can overpower the n-channel transistors in the pull-down network and the small inverter whose output is connected to x. Thus, node x is driven high. Once x is high, node y is driven low. At the end of the precharge phase, the output, y, of the gate is low.

The second phase is the *evaluate* phase. In this phase, the precharge-bar node, pb, is set-high, and the logic inputs are allowed to change to high. If the combination of high inputs establishes a connection from node x to ground, then node x goes low, which causes node y to be driven high. If the logic function is not satisfied, then the small feedback inverter maintains the high level on node x and node y remains low. At the end of the evaluate phase, the output of the gate is the value of the function for the current inputs.

For example, the pull-down network of the AND-OR gate makes a connection from x to ground if the input a is high and if either the b or c input is high. Thus, this gate computes the function

$$a \wedge (b \vee c).$$

The third phase is the *hold* phase. In this phase, the inputs return to low values, and the precharge-bar node, pb, remains high. In this phase, the pull-down network does not make a

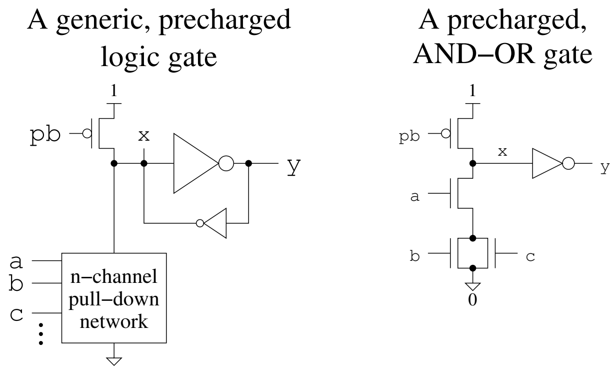
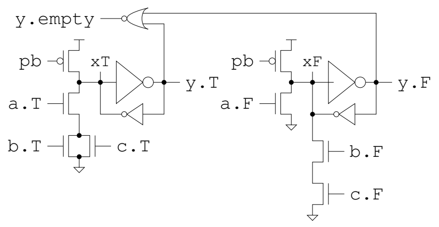

A generic, precharged logic gate
1
pb
x
y
a
b
c
n–channel pull–down network
A precharged, AND–OR gate
1
pb
x
y
a
b
c
0

Fig. 10. Precharged logic gates



Dual–Rail Code

y.T	y.F	*y*
L	L	empty
L	H	false
H	L	true
H	H	invalid

Fig. 11. A dual-rail AND-OR gate

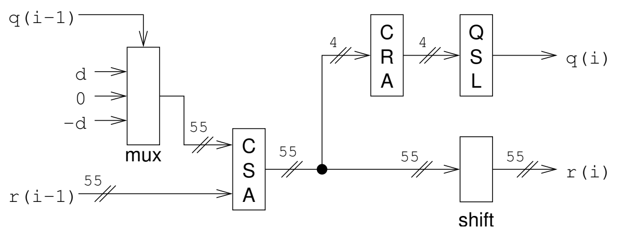


Fig. 12. A Divider Stage

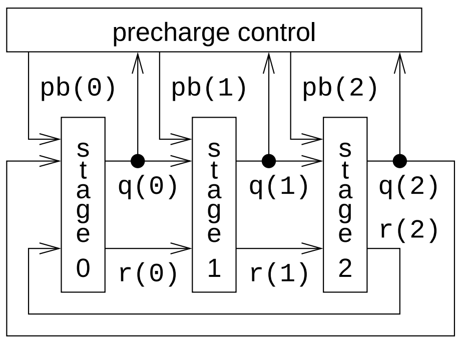


Fig. 13. Divider Architecture

connection from node x to ground, and the pull-up transistor does not make a connection from x to power. In this phase, the small feedback inverter maintains node x at the value that it had at the end of the evaluate phase. Throughout the hold phase, the output of the gate is the value of the function from the preceding evaluate phase.

The divider that we verified is self-timed. Rather than using a clock signal to control the sequencing in the SRT iterations, control circuitry detects when stages complete their operations and enables subsequent operations accordingly. To enable completion detection, most signals are encoded using a *dual-rail code* [43]. If y is a boolean valued variable, then node y.T is driven high during an evaluate phase if the function for y evaluates to true, and y.F is driven high if the function for y evaluates to false. During the precharge phase, both y.T and y.F are driven low. Therefore, the completion of precharge is indicated when both y.T and y.F are low, and the completion of evaluation is indicated when either y.T and y.F are high. Figure 11 shows a dual-rail AND-OR gate with completion detection.

4.4 The Divider Chip

Figure 12 shows the hardware that implements a single step of the SRT division algorithm. Stage i receives the partial remainder and a quotient digit from stage i-1 on buses r(i-1) and q(i-1) respectively. The quotient digit, q(i-1), is used to select whether d, 0, or -d will be added in the carry-save adder, CSA, to the previous partial remainder, r(i-1), where d is the divisor (see section 4.2). This shift module is simply a relabeling of wires that shifts the output of the carry-save adder one bit to the left to effect a multiply by two. The carry-ripple adder, CRA, converts the top four bits of the partial remainder to the standard, non-redundant representation. The quotient-select-logic, QSL, uses this sum to determine the next quotient digit.

As depicted in figure 13, the entire design consists of three of the above stages, arranged in a ring [51]. As mentioned in section 4.3, the iteration is controlled by embedding completion information in the data signals. Most values are encoded using dual-rail codes. Quotient bits are encoded using a three-wire, "one-hot" scheme [30]. At most one wire may be high at any time; the four values of the code are empty, -1, 0, and +1.

The divider design includes two performance optimizations that exploit timing details of the circuits. First, a stage can precharge faster than its predecessor can evaluate. The 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. Second, the partial remainder word becomes valid during an evaluate phase before the quotient digit, relies on the quotient digit of a stage being the last output to change during the evaluation phase. This allows the computation of stage i+1 to start as soon as the quotient digit from stage i is output, without any additional hardware to check the completion status of the partial remainder.

5 Verification Strategy

Our overall goal is to verify that the transistor-level implementation of William's divider correctly computes division. There are two major challenges to be faced: First, we have to show that the algorithm that this design is based on is functionally correct. Second, we need to demonstrate that the implementation adheres to this algorithm. In particular, this requires showing that the timing-critical control of the self-timed design works as intended.

Since the division algorithm is of an iterative nature, we employed an invariant-based argument to establish its correctness. However, it is not practical to directly formulate the necessary invariants at the level of detail for which we verify the divider: a timed, transistor level model. In particular, such an invariant would be too tedious for manual formulation; it is unreasonable to expect state-space-exploration-based tools to automatically find an invariant that supports the property of correct division; and the heuristics of sophisticated theorem provers would almost certainly get lost in the myriad of details of the model without finding the desired proof.

Having concluded that we cannot hope to directly verify the correctness of the detailed model, we pursued an approach that relies heavily on refinement to manage the complexity of the proof. Our goal is to prove a safety-property of the design; this permits the use of refinement to 'inherit' correctness properties that are shown for an abstraction of the design down to the implementation level [15]. We have devised a hierarchy of refinements that relates an abstract, algorithmic descriptions of SRT division on rational numbers to the transistor-level implementation of the design. There are several intermediate descriptions in the hierarchy; these have been chosen to allow each successive refinement step to focus on one particular aspect of the overall correctness property.

The transistor-level implementation contains thousands of *timed* signals. This makes both deductive and state-exploration-based reasoning about the timed behavior of the design impractical. Instead, we employ a refinement argument between the transistor-level implementation and the next higher model in our refinement hierarchy. This argument requires us to show that certain timing properties hold for the transistor level model. These can be shown by simple graph-traversal methods given that certain side conditions hold. In our framework, the graph-traversal algorithm is incorporated as a domain-specific decision procedure, and the side conditions become proof obligations that we discharge using deductive, refinement based arguments. By combining domain-specific decision procedures with general purpose deductive theorem proving, we can verify a design that would not be possible or practical to verify by either approach alone.

In the following, we first introduce theorems that provide the foundation for refinement-based reasoning about ST programs. We then give an overview of the refinement hierarchy used to show correctness of the divider, and finally provide more detailed descriptions of each proof step.

5.1 Refinement between ST Programs

Intuitively, program P' is a refinement of P if every state transition that P' can make corresponds to a move of P. More formally, let S' be the state space of P', S be the state space of P, and A be a mapping from S' to S. The mapping A is referred to as an *abstraction mapping*; for simplicity, we assume that A is a function. Let Q_0 and T be the initial state predicate and set of transitions for P and Q'_0, T' the same for P'.

Let s'_1 be a state of P'. We say that the transitions of P' are matched by the transitions of P at state s'_1 *iff* for every state s'_2 that is reachable by performing a single transition of P', there is a transition of P that effects a move from $A(s'_1)$ to $A(s'_2)$. Stuttering actions (where $A(s'_1) = A(s'_2)$) are exempted. We write $match_{trans,A}(T',T)(s')$ to denote that the transitions of P' are matched by those of P at state s', with

$$\begin{aligned}
match_{trans,A}(T',T)(s') \quad &= \\
\forall \ll G' \rightarrow M' \gg \in T'.\ & G'(s') \Rightarrow \\
& (A \circ M')(s') = A(s') \\
& \vee\ \exists \ll G \rightarrow M \gg \in T. \\
& \qquad (G \circ A)(s') \\
& \qquad \wedge\ (A \circ M')(s') = (M \circ A)(s')
\end{aligned}$$

where M and M' denote multi-assignments of program P and P' respectively. Note that a given transition $t' \in T'$ does not necessarily have to be matched with a *single* transition $t \in T$. Instead, the matching can be subject to the particular state s' of P'; for instance, t' may be matched with t_1 if P' is in state s'_1 and with t_2 if P' is in state s'_2. In the theorem prover, such case-splitting is straightforward.

Both P' and P may have associated protocols, $proto$ and $proto'$. A protocol is a relation on the states of a program, e.g.,

$$proto \subseteq S \times S$$

We use protocols to model assumptions about the environment of a model; a transition from state s_1 to s_2 is admitted if $(s_1, s_2) \in proto$. In the context of the divider verification, we use protocols to constrain the passage of time in timed models (see section 6.5).

We define $match_{proto,A}$ in a similar fashion to indicate matching of protocol actions, again exempting stuttering actions:

$$\begin{aligned}
match_{proto,A}(proto', proto)(s'_1) \quad &= \\
\forall s'_2 \in S'.\ & \\
& proto'(s'_1, s'_2) \Rightarrow proto(A(s'_1), A(s'_2)) \\
& \vee\ A(s'_1) = A(s'_2)
\end{aligned}$$

All actions of P' are matched by P at state s' if both transition actions and protocol actions are matched:

$$\begin{aligned}
Q_{P' \preceq_A P}(s') = \quad & match_{trans,A}(T',T)(s') \\
& \wedge\ match_{proto,A}(proto', proto)(s')
\end{aligned}$$

Noting that $Q_{P' \preceq_A P}(s')$ is a predicate over states of P', we say that P' is a refinement of P under abstraction mapping A if $Q'_0 \Rightarrow Q_0 \circ A$ and $Q_{P' \preceq_A P}(s')$ is a safety property of P'. We write $P' \preceq_A P$ to denote this. Because refinement is a safety property, refinement for Synchronized Transitions programs can be defined using our *wp* semantics presented in section 1.3. As shown below, verification of refinement can often be reduced to a simple problem of tautology checking.

Note that any interpretation of refinement as a 'correctness property' in the sense of stating that 'P is correct because it is shown to be a refinement of the specification P''

is subject to the abstraction mapping faithfully capturing the intention of the person who defined it. In particular, an abstraction mapping may in fact be defined such that it performs part of the computation that the implementation is expected to do. Furthermore, our notion of refinement does not consider liveness properties; an implementation that does nothing can always be shown to be a refinement of a given specification (since it will never perform an action that violates the specification). In either case, such an abstraction mapping would clearly not capture the verifier's intention.

Avoiding such pitfalls requires careful construction of abstraction mappings. In our refinement proofs, we have whenever possible used abstraction mappings that are composed from mappings between state variables of the implementation and the specification which are easily seen to correspond to what was intended. For example, one such mapping is a function that maps a vector of Boolean variables representing an integer in two's complement notation into its specification-level interpretation as a signed integer.

The next two theorems formalize two useful relationships between refinement and safety properties. The first theorem states that safety properties of a more abstract program are inherited by refinements of the program.

Theorem 1. *Given programs P and P', an abstraction function A such that $P' \preceq_A P$, and a predicate Q such that Q is a safety property of P. Then $Q \circ A$ is a safety property of P'.*

This theorem is easily proven by induction over traces of P'.

The second theorem describes how safety properties of P can be used to show that P' is a refinement. This often reduces the problem of showing refinement to one of automatic tautology checking.

Theorem 2. *Given programs P and P' with initial state predicates Q_0 and Q'_0, an abstraction function A, and a predicate Q such that Q is a safety property of P. If $Q'_0 \Rightarrow Q_0 \circ A$ and $(Q \circ A) \Rightarrow Q_{P' \preceq_A P}$, then $P' \preceq_A P$.*

A simple induction argument over traces of P' shows that $Q_{P' \preceq_A P}(s')$ is a safety property of P' and establishes the claim.

Based on these theorems, we have implemented proof rules for refinement between ST programs. For instance, the rule corresponding to theorem 2 reduces an obligation of the form

$$\mathit{IsRefinement}(Q'_0, P', \mathit{proto}', Q_0, P, \mathit{proto}, A)$$

to the obligation

$$\begin{aligned} & Q'_0 \Rightarrow Q_0 \circ A \\ \wedge\ & (Q \circ A) \Rightarrow Q_{P' \preceq_A P} \\ \wedge\ & \mathit{HasSafetyProperty}(Q_0, P, \mathit{proto}, Q) \end{aligned}$$

Note that

$$(Q'_0 \Rightarrow Q_0 \circ A) \wedge ((Q \circ A) \Rightarrow Q_{P' \preceq_A P})$$

is a simple predicate over states of P'. The only invariant that we must find is the one required for

$$\mathit{HasSafetyProperty}(Q_0, P, \mathit{proto}, Q)$$

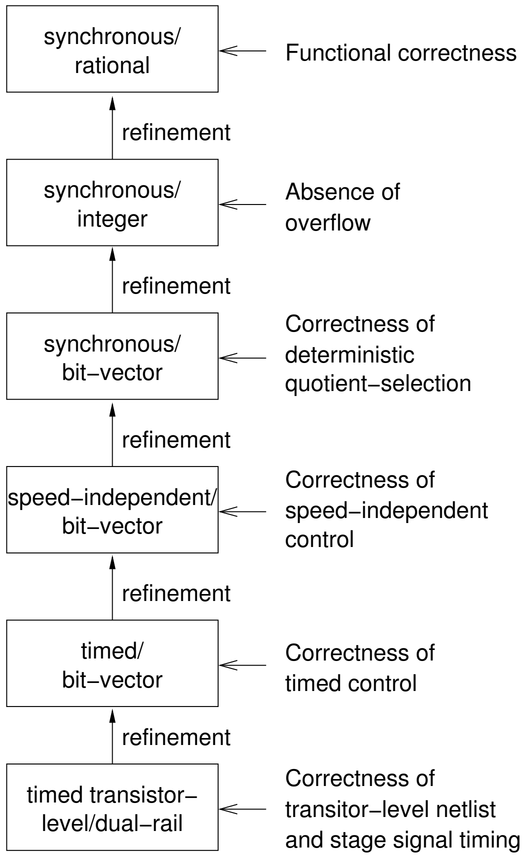

synchronous/ rational
Functional correctness
refinement
synchronous/ integer
Absence of overflow
refinement
synchronous/ bit−vector
Correctness of deterministic quotient−selection
refinement
speed−independent/ bit−vector
Correctness of speed−independent control
refinement
timed/ bit−vector
Correctness of timed control
refinement
timed transistor−level/dual−rail
Correctness of transitor−level netlist and stage signal timing

Fig. 14. Refinement Hierarchy

In other words, we use an invariant of P to establish that P' refines P and therefore P' inherits the safety properties of P. By carefully designing a refinement hierarchy, we can dramatically simplify the task of finding the invariants needed for verification.

In our implementation, the abstraction mapping A is given in the form of a list of equations $(v_1 = e'_1, \ldots, v_k = e'_k)$, where $v_1, \ldots, v_k$ are variables of the specification P, and $e'_1, \ldots, e'_k$ are expressions on the variables of P'. The rule constructs expressions for $(Q \circ A)$ and $Q_{P' \preceq_A P}$ by analyzing the syntactic structure of P' and P, and appropriately combining subexpressions of the programs. Expressions corresponding to function composition are computed using substitution; this is possible since both multi-assignments and the abstraction mapping are represented as suitable functional vectors.

5.2 *Refinement Hierarchy for the Divider*

Figure 14 depicts the hierarchy of models that we used to verify the divider. Each model is named based on a characterization of the type of control and the data representation used. For instance, the synchronous/bit-vector model has synchronous control and uses a bit-vector representation for the divisor and partial remainder. For each level of abstraction, the figure indicates the aspect of overall correctness that the corresponding refinement step focuses on.

The first two refinement steps are data refinements. Our top-level model has a single stage which computes a quotient digit 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 the self-timed 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; these bounds are exploited for an optimization of the handshaking protocol.

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 signal is set independently. Furthermore, the status of the quotient bit is used to determine the status of the entire output of the stage. This optimization leads to a smaller, faster implementation, but it also introduces timing dependencies that we must verify.

6 Verifying a Self-Timed Divider: The Proofs

This section presents each of the refinement proofs for the divider in detail. We also present the safety and refinement properties that are established for each model along with the proofs for these properties. This shows how we can model a significant design at many different levels of abstraction, and how proofs can be carried out using a combination of generic deductive arguments and domain specific decision procedures.

We present the proofs in a top-down fashion. This choice is both historical and pedagogical. The top-down order reflects the order in which we developed the proofs. Furthermore, just as the proofs at the higher levels of the hierarchy provide invariants needed by the lower level proofs, the descriptions of the higher level proofs provide a context in which to understand the lower level ones. A consequence of this presentation is that some of the most novel features of our approach, such as reasoning about timed, switch-level models, are presented later in this section.

6.1 Functional Correctness of the Synchronous/Rational Model

The top-level model in our refinement hierarchy is an ST program that performs SRT division on rational numbers. We begin our verification by proving that this program indeed correctly divides.

Figure 15 depicts the ST code of the quotient selection logic of our top-level, synchronous divider model. The logic is modeled as three ST transitions which are enclosed in a cell. A cell is a ST construct that permits the encapsulation and re-use of ST code. Cells are instantiated much like macros; instantiating a cell has the effect of creating a copy of the transitions in the cell, with the cell's formal parameters replaced by the actual parameters of the instantiation.

Function `nextRem` computes the remainder of the next iteration as

$$\begin{aligned}&\mathtt{nextRemF}(\mathtt{rem}, \mathtt{quot}, \mathtt{divisor})\\ &\quad = 2 \cdot \mathtt{rem} - \mathtt{quot} \cdot \mathtt{divisor}\end{aligned}$$

As described in section 4.2, radix-2 SRT allows quotient digits to have the value -1, 0 or 1. 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 highest-level description of the divider this overlapping choice for the digit is represented by three transitions combined with the asynchronous combinator (see fig. 15). If more that one transition is enabled, a non-deterministic choice takes place. 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 non-determinism, we avoid cluttering this description with implementation details, and at the same time modularize and simplify the proofs. Deterministic quotient digit selection is introduced in the synchronous, bit-vector model.

The complete program includes additional, synchronously composed transitions that instantiate the cell `SRTDivide`, store the computed quotient digits in a vector, and handle the loading of operands and the generation of completion signals. This logic also ensures that the program's external behavior appears synchronous despite the use of the asynchronous combinator, which is employed solely to express the internal non-determinism.

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

$$|R_i| \leq D \tag{1}$$

$$2C - D\sum_{j=0}^{i-1} q_j 2^{-j} = R_i 2^{1-i}, \tag{2}$$

where R_i is the remainder determined in iteration i, and $q_0, \ldots, q_{i-1}$ are the quotient digits computed so far. From these two invariants and the initial condition that the divisor D and dividend C are normalized to satisfy $\frac{1}{2} \leq D < 1$ and $0 < C < D$, we can prove that

$$\begin{aligned}&\mathit{DuringComputation} \Rightarrow\\ &\quad \left|2\tfrac{C}{D} - \textstyle\sum_{j=0}^{i-1} q_j 2^{-j}\right| \leq 2^{-(i-1)}\end{aligned} \tag{3}$$

is a safety property of ST program `srtDivider`. *DuringComputation* is a predicate stating that the divider is properly initialized and currently computing a quotient. This result demonstrates that the computed quotient asymptotically approaches the true quotient C/D.

```
IMPLEMENTATION MODULE srtDivider;
  STATIC
    (* ... *)

    SRTDivide : SRTDivideC = (* CELL( quot: sInt2; rem: Rational;
                                      STATIC divisor: Rational; *)
    BEGIN
        <<  nextRem(rem,quot,divisor) <= 0.0
            -> rem, quot := nextRem(rem,quot,divisor), -1 >>
     || <<  (-divisor <= 2.0*nextRem(rem,quot,divisor)) AND
            (2.0*nextRem(rem,quot,divisor) <= divisor)
            -> rem, quot := nextRem(rem,quot,divisor), 0 >>
     || <<  0.0 <= nextRem(rem,quot,divisor)
            -> rem, quot := nextRem(rem,quot,divisor), 1 >>
    END;

    (* ... *)
END.
```

Fig. 15. Quotient Selection in Functional Model

Proving invariants (1) and (2) requires a few additional book-keeping invariants that e.g. state that the quotient digit always is one of $-1, 0, 1$. Together, these invariants are easily proved by case-split over the value of the quotient digit, rewriting with axioms about summation and powers-of-two, and application of the decision procedure for linear arithmetic. Using the these invariants, the safety property (3) is proved in a similar fashion.

As an example, figure 16 depicts the complete proof for invariant (1). Tactic `caseQuot` is a user-defined tactic that performs a case-split over the possible valuations for the quotient digit, and is defined as follows:

```
val caseQuot =
    caseSplit "quot"
          (map (fn c => (Not init %=> c))
                [(quot %= (## (~1))),
                 (quot %= (## 0)),
                 (quot %= (## 1))])
```

The operators `%=>`, `%=`, and `##`, denote implication, test for equality, and construction of a constant in the object logic. The distinct symbols allow the user to use both the built-in ML operators and the operators in the object logic when constructing a proof. Tactic `caseSplit` is the core tactic for case splits, and is specialized into the case-split over quotient digit using SML-level partial evaluation.

The proof in figure 16 is written using the goal package; this allows steps of this proof to be cut-and-pasted into an interactive proof session. Function `TAC` is part of the goal-package and applies a given tactic to the current proof state. Function `mapTac` applies a given tactickle to selected obligations of a proof state. For instance, the last proof step applies the arithmetic decision procedure to all obligations, thus discharging all pending obligations.

6.2 Synchronous/Integer Model

In the top layers of refinement, we focus on transforming the data-path of the circuit from one expressed in terms of rational numbers to an implementation in terms of bit-vectors of length M.

In preparation for the refinement to bit-vectors, we first introduce an intermediate model, `srtIntDivider`. This model operates on (signed) integers in the range $\{-2^{M-1}, \ldots, 2^{M-1} - 1\}$, instead of rational numbers. Correspondingly, all operations on rational numbers are replaced by operations on signed integers, modulo (2^N).

For instance, in `srtIntDivider`, the ST expression `2.0 * rem - divisor` is correspondingly written as

```
plusMod2N(M, rem,
    minusMod2N(M, rem, divisor))
```

In this expression, `plusMod2N` is a function that is defined to have the exact same semantics as (signed) addition on M-bit, two's complement, bit-vectors. This property will greatly simplify the next-lower refinement step.

The abstraction mapping between `srtIntDivider` and `srtDivider` divides the `rem`, `divisor` and `dividend` variables of `srtIntDivider` by 2^{M-1}, and directly maps all other variables to their counterpart in `srtDivider` (variables of `srtIntDivider` are primed):

$$A_{IntDiv} : \begin{pmatrix} \texttt{rem} & \leftarrow & 2^{(1-M)}\texttt{rem}' \\ \texttt{divisor} & \leftarrow & 2^{(1-M)}\texttt{divisor}' \\ \texttt{dividend} & \leftarrow & 2^{(1-M)}\texttt{dividend}' \\ \texttt{quot} & \leftarrow & \texttt{dividend}' \\ & \vdots & \\ \texttt{count} & \leftarrow & \texttt{count}' \end{pmatrix}$$

In order to prove refinement, it is necessary to show that there are no overflows in the modulo-2^N arithmetic performed by `srtIntDivider`. However, this property is implied by the invariants of `srtDivider` that were already used in the

```
(* proof for clmInvAbsrem_lt_divisor *)
fun prfInvAbsrem_lt_divisor ps = (
    setGoal ps; setObl (Idx 0); (* set up goal package *)

    tac setupInv;  (* apply the invariant-by-WP rule *)
    tac caseQuot;  (* case-split over quotient digit *)

    (* split by transition and move guard to assumptions *)
    TAC (mapTac (tryTac conjSplitAll) (Nms "_.case_quot"));
    TAC (mapTac (tryTac impReduction) (Nms "_.case_quot"));

    (* use the guard to simplify the case assumptions, and substitute them
       throughout the obligation, then simplify *)
    TAC (mapTac (tryMapSeq unCond (sANms "case_quot"))
                (Nms "_.case_quot"));
    TAC (mapTac (eqRewrite (sANms "case_quot", sAll))
                (Nms "_.case_quot"));
    TAC (mapTac (simpIfsAll thenTac liftIfsAll thenTac
                 simpIntAll thenTac simpRatAll)
                (Nms "_.case_quot"));

    (* split obligations by invariant clause *)
    TAC (mapTac (tryTac conjSplitAll) (Nms "_.case_quot"));

    (* discharge obligations *)
    TAC (mapTac linArith All);

    getGoal())
```

Fig. 16. Proof for invariant (1)

functional correctness proof, plus an additional invariant of `srtDivider` which constrains the range of the current remainder, given that the current quotient has a particular value. Thus, applying theorem 2, we can prove refinement by showing that these invariants (after being subjected to the abstraction mapping A_{IntDiv}) together imply the refinement predicate $Q_{\texttt{srtIntDivider} \preceq_A \texttt{srtDivider}}$[2].

The proof that this implication indeed holds proceeds by case-split over the value of the quotient digit, then requires a non-trivial amount of rewriting with lemmas about the modulo-2^N operations, powers of two, etc., and then uses the decision procedure for linear arithmetic to discharge obligations.

The amount of rewriting required made this proof one of the least automated of the entire verification; a certain degree of automation was achieved using proof tactics that automatically apply a set of lemmas as rewrite-rules.

6.3 Synchronous/Bit-Vector Model

In the next refinement step, we further refine the data-path to use a bit-vector representation for the divisor and current remainder: In the synchronous/bit-vector model, the remainder is maintained in carry-save representation. Furthermore, the next quotient digit is computed deterministically based on the top four bits of the carry-save adder. Only the top four bits of the carry and save parts of the remainder are resolved by a carry-ripple adder and fed to the quotient selection logic; the carry of the bottom bits does not need to be resolved.

Figure 17 shows the transitions of the quotient selection logic. `CRASumBit` is a function that computes the value of bit `b` $\in \{0, \ldots, 3\}$ of the output of the carry-ripple adder, depending on the current remainder and previous quotient digit. Depending on these bits, the next quotient is set to 1, 0 or -1, in two-bit signed integer representation.

The abstraction mapping between the synchronous/bit-vector model and the synchronous/integer model is written in terms of a function `bitVec2sIntN` which interprets a vector of booleans as a signed integer in 2's complement representation. This function is used to map the (M-bit) divisor and (2-bit) quotient of the synchronous/bit-vector model into the corresponding integer-subrange variables of the synchronous/integer model. Furthermore, the mapping for the remainder uses the signed modulo-2^N arithmetic functions to consolidate the carry and save portions of the remainder at the bit-vector level into the remainder at the integer level.

In order to prove refinement between the two models, 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. More precisely, this requires that the quotient choice made by the bit-vector model falls within a region that allows the integer model to make a consistent choice. Note that for a given current remainder and quotient digit, the integer model may have more than one choice for the next

[2] We also need to show that the invariants involved are in fact safety properties, which is trivial.

```
QSL : QSLC = (* CELL(outquotient: Word2; sum, carry: WordN;
                     quotient: Word2; divisor: WordN); *)
TYPE bFn = FUNCTION(b:INTEGER): BOOLEAN;
STATIC
    cra : bFn = BEGIN
                    CRASumBit(carry, sum, divisor, quotient, b)
                END;
BEGIN
      << outquotient(0) :=    (NOT cra(2))
                           OR (cra(2) AND (NOT (cra(1) AND cra(0))))   >>
    * << outquotient(1) :=    (cra(2) AND (NOT (cra(1) AND cra(0))))
                           OR ((NOT cra(2)) AND cra(3) AND cra(1) AND cra(0)) >>
END;
```

Fig. 17. Quotient Selection in Bit-Vector Model

quotient digit; all that is required for the implementation is to choose a quotient digit that is consistent with *one* of these choices.

Similar to the previous refinement step, the refinement predicate is implied by safety properties inherited from the synchronous/integer model, which were in turn inherited from the top-level specification. This allows us again to prove refinement without directly establishing invariants of the synchronous/bit-vector model. The inherited properties establish constraints on the possible combinations of values of the divider, current partial remainder and current quotient digit at each iteration. One such property states that if the current quotient is -1, then the remainder cannot be positive. This is straightforward to observe, and prove, at the functional level; figure 15 shows that the transition that sets the quotient to -1 is enabled only if the next remainder is non-positive. Directly formulating and proving this property at the level of detail provided by the bit-vector-level model would have been much more tedious.

The proof that the inherited safety properties imply the refinement predicate requires reasoning about bit-vectors interpreted as signed integers, and operations thereon. A high degree of automation for this proof step is achieved by using a BDD-based rewrite rule for bit-vector arithmetic. This proof rule 'interprets' the function `bitVec2sIntN` as well as the modulo-2^N operations such as `plusMod2N` and rewrites expressions written in terms of these functions into an equivalent boolean expression, represented as a BDD.

Using this decision procedure, the refinement proof is very concise. After case splitting over the value of the quotient digit, each of these cases is automatically discharged by the following proof tactic:

```
val dischBV =         rewriteBVAll
              thenTac liftIfsAll
              thenTac rewriteBVAll
              thenTac predCalcDisch;
```

This tactic first applies the bit-vector rewrite procedure, then lifts If expressions, applies the rewrite procedure again, and then uses the BDD-based decision procedure for propositional logic to show that the resulting obligation is a tautology. Using this tactic, the complete proof can be written as

```
(* split cases over quotient digit *)
tac caseQuot;

(* re-write with quotient digit & simplify *)
TAC (mapTac
      (          (splitAnt (sANm "case_quot"))
         thenTac (eqRewrite
                     (sANms "case_quot", sAll))
         thenTac simpIfsAll)
      (Nms "_.case_quot"));

(* discharge the cases *)
TAC (mapTac dischBV All);
```

When we first attempted this proof, we discovered quotient selections that could lead to a violation on the 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. As previously indicated, we reconstructed our models from published descriptions [50, 51]. Based on our interpretation of these sources, we originally designed the divider with a quotient-selection logic that is based on a three-bit carry-ripple adder. However, doing so causes an incorrect quotient choice under certain circumstance involving overflow of the carry-ripple addition. We modified the design to use a fourth bit in the quotient select logic as described in section 4.2. We were then able to prove refinement for the modified design. Later on, communication with the original designer confirmed that our initial implementation was incomplete. Although we did not find an error in the actual design, our verification effort revealed a place where the published documentation for the design was unclear.

6.4 Speed-Independent/Bit-Vector Model

In the following two refinement steps, we focus on the control logic of the design, while the data-path remains unchanged (i.e., the next two levels use exactly the same representation and computation for the current remainder and quotient digit as the synchronous/bit-vector model).

In the first of the two refinement steps, the synchronous/bit-vector model is refined into the speed-independent/bit-vector model. Instead of computing a new remainder and quotient

```
Process: ProcessCell = (* CELL(STATIC stage: INTEGER; pb, q: ABool; sum,carry: AWordN;
                               cpaSum: AWordN; quotient: AWordN; divisor: WordN); *)
STATIC
    prev: INTEGER = (stage+2) MOD 3;
    next: INTEGER = (stage+1) MOD 3;
    me: INTEGER = stage;
BEGIN
    (* if successor stage has finished eval and is holding the output,
            we can start precharging *)
        << pb(next) AND q(next) AND pb(me) -> pb(me) := FALSE >>
        * ClearStage(me, sum, carry, cpaSum, quotient)

    (* if we're done precharging, we can drop the done signal *)
    ||  << NOT pb(me) AND q(me) -> q(me) := FALSE >>

    (* enable evaluation, if the successor has started precharging *)
    ||  << NOT pb(next) AND NOT pb(me) -> pb(me) := TRUE >>

    (* when successor has finished precharging, we can output the next quotient *)
    ||      <<  pb(me) AND (NOT q(next)) AND NOT q(me) -> q(me) := TRUE >>
         *  Divide(me, prev, sum, carry, cpaSum, quotient, divisor)
END;
```

Fig. 18. Control Logic of Speed-Independent/Bit-Vector Model

digit in every iteration, this model mimics the flow of control and data of the implementation: There are three 'stages' which cycle through states corresponding to the precharge, evaluate and hold phases of the implementation stages. When a stage is precharging, all of its outputs and internal signals are forced to empty values. When a stage is evaluating, it computes new valid (i.e. non-empty) values for its outputs based on the values of its inputs. When a stage is holding, its outputs retain the values from the preceding evaluate phase; a holding stage provides stable inputs to its successor which is evaluating.

Because the stages of the divider are connected in a ring, one might expect to find cyclic dependencies on data values. A fundamental safety property of the divider design is that at all times, at least one stage is in the precharge state. Because the signals of this stage are forced to empty values regardless of the values of the inputs, the precharge stage provides a cut in the data-value dependency graph. This breaking of cyclic dependencies greatly simplifies analysis of the divider.

At this level of the refinement hierarchy, we model all communication in the control logic and the data path as being speed-independent. This means that the computation that is eventually performed is independent of the delays of the components. Speed independence is a safety property [29]: there must never be two transitions simultaneously enabled such that one can modify the value of a variable read by the other. By using a speed-independent model at this level of the hierarchy, we establish important invariants without the complications that arise when reasoning about timed models.

Figure 18 shows the ST cell that implements the control of the speed-independent/bit-vector model. This cell is instantiated once for each of the three stages. The control state of a stage i is determined by the 'precharge-bar' and completion signals $\mathtt{pb}(i)$ and $\mathtt{q}(i)$, respectively. For instance, if $\mathtt{pb}(i) = \mathit{true}$ and $\mathtt{q}(i) = \mathit{false}$, then stage i is in evaluation mode.

Figure 19 illustrates the flow of control as governed by the speed-independent protocol. The upper line of three characters in each state label gives the value (high or low) of `pb(0)`, `pb(1)`, and `pb(2)`. The lower line gives the state of the outputs of stages 0, 1, and 2 (empty or valid).

The actual computation of the next quotient is performed by ST cell `Divide`; it is identical to the computation performed by the data-path of the synchronous/bit-vector program (in fact, both programs import `Divide` from the same ST source module). Cell `ClearStage` mimics the effect of pre-charging a stage by forcing all outputs to a known value (in this case all zeros). This substantially simplifies the construction of an abstraction mapping during the last refinement step to the transistor-level implementation.

Intuitively, the abstraction mapping from the speed-independent model to the synchronous one needs to track the circular flow of data through the ring formed by the three stages, and pick the values of the quotient and partial remainder out of the stage that currently holds valid values on its outputs. The mapping is constructed using two predicates,

$$\begin{aligned}\mathtt{holdModeP}(i) &= \mathtt{pb}(i) \land \mathtt{q}(i)\\ \mathtt{holdsDataP}(i) &= \mathtt{holdModeP}(i)\\ &\qquad \land \neg\mathtt{holdModeP}((i+1) \bmod 3)\end{aligned}$$

Predicate `holdModeP` holds for a stage that is in hold mode. However, the speed-independent protocol allows a state where two successive stages both satisfy `holdModeP`. In this case, the first stage is enabled to begin precharging and may destroy its outputs at any time. Therefore, predicate `holdsDataP` which characterizes the stage that holds

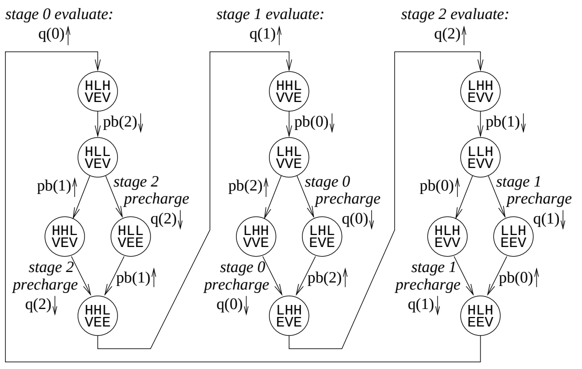


Fig. 19. Speed-Independent Protocol

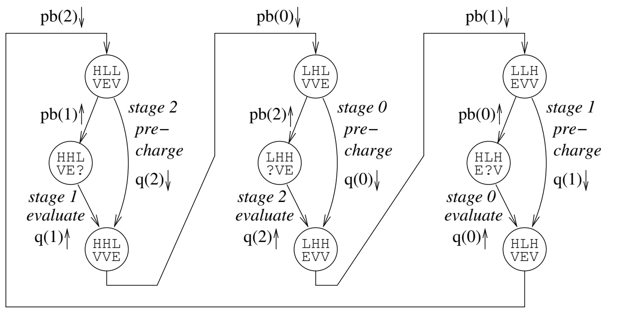


Fig. 20. A Timed Implementation

the current quotient and partial remainder requires the clause $\neg$holdModeP$((i+1)\text{mod}3)$. This ensures that the values are taken from the stage that is indeed holding its outputs while its successor is evaluating.

The abstraction mapping itself is then written as a case-statement which maps the outputs of the stage that satisfies holdsDataP to the corresponding variables of the synchronous model.

Proving refinement entails showing that the speed-independent control protocol works correctly, and that the stage identified by holdsDataP is indeed the one that holds the correct values for the quotient digit and partial remainder of the current iteration of the SRT algorithm. Note that the expressions that actually compute these values are identical in both programs. Therefore, discharging obligations that state the equivalence of the computations performed by both programs is trivial.

Correct operation of the speed-independent control is established by way of an invariant that enumerates reachable control states as depicted in figure 19. Since there are only 15 such states, the invariant was easily formulated. Proving that this invariant as well as the refinement predicate are safety-properties was then done completely automatically using the BDD-based decision procedure for propositional logic.

6.5 Timed/Bit-Vector Model

In the speed-independent model, the precharge control logic performs an explicit check to ensure that stage i+1 has finished precharging before stage i sets its outputs. This corresponds to the clause NOT q(next) in the guard of the last transition in figure 18. In the timed model, the control logic only tests the completion of evaluation, and timing bounds are used to ensure that the precharging in stage i+1 completes before the evaluation in stage i. This corresponds to Williams' first optimization in the design of the chip, as discussed in section 4.4. Figure 20 depicts the corresponding state transition diagram.

We use the approach of [3] to model time: a real-valued variable tau is added to the program to model the current time, transition guards are strengthened to express lower bounds on delays, and an action for advancing time is defined in the form of an environment protocol so as to observe upper bounds on delays (i.e. time may not progress beyond the maximum delay for a pending action).

A technical issue that must be addressed is that of "Zenoness" (see [3]). In particular, from any reachable state, the model must admit an execution where time advances without bound. For models such as those presented in this paper, this property can be established syntactically: (1) the model has a finite set of possible upper bounds for time; (2) when any bound is active, there is an enabled transition that can be performed to remove the bound; (3) there is a constant, $\epsilon > 0$ such that when any upper bound becomes active, it is for a time at least ϵ time units in the future. Currently, we have been satisfied with verifying these properties by inspection of the program. In principle, these tests could be automated, but we have not seen a need to do so.

In the timed/bit-vector model, the variables pb(i) and q(i) are records with two components. The first component, v, holds the actual value, while the second, t maintains the time at which the variable was last updated.

The clause NOT q(next) of the guard for the evaluate action that asserts that the successor stage is done precharging is replaced by a clause that states that the successor stage started precharging sufficiently far in the past. Thus, the last transition in figure 18 is replaced by

```
    <<     pb(me).v
       AND (tau >= pb(me).t + MinEvalTime)
       AND NOT q(me).v
           -> q(me).v, q(me).t  := TRUE, tau >>
  * Divide(me, prev, sum, carry, cpaSum,
           quotient, divisor)
```

MinEvalTime is a constant that provides a lower bound on the time a stage takes to compute a new quotient.

In addition, an upper bound on the time taken to precharge a stage is asserted by a protocol

$$\begin{aligned}&\neg \mathit{Pre}\{\texttt{pb(me).v}\} \wedge \mathit{Pre}\{\texttt{q(me).v}\}\\ \Rightarrow\ &\mathit{Post}\{\texttt{tau}\} \le \mathit{Post}\{\texttt{pb(me).t}\} + \texttt{MaxPrechargeTime}\end{aligned}$$

Recall that a protocol is a predicate that defines a relation from states to states which poses a constraint on allowed state transitions. The Pre and Post attributes of the state variables are used to distinguish between variables in the pre- and the post-state of the environment action.

The above protocol states that whenever the second transition of figure 18 (whose execution marks the end of the precharge phase of a stage) is enabled, time has not progressed more than `MaxPrechargeTime` time units since precharging has begun. In other words, precharging never takes longer than `MaxPrechargeTime`.

The abstraction mapping from the timed/bit-vector model to the speed-independent/bit-vector one is simple: The `.v` components of the timed variables are mapped to their counterparts in the higher-level model, `tau` and the `.t` components are dropped, and all other variables are mapped directly.

To prove refinement between the two models, it is necessary to show that whenever the evaluate transition of the timed model is enabled, so is the corresponding transition of the synchronous model. This corresponds to showing that for each stage `me`,

$$\mathtt{pb(me).v} \wedge \mathtt{tau} \geq \mathtt{pb(me).t} + \mathtt{MinEvalTime} \wedge \neg\mathtt{q(me).v} \\ \Rightarrow \neg\mathtt{q(next).v}$$

is a safety property of the timed model.

Using the assumption

$$\mathtt{MaxPrechargeTime} < \mathtt{MinEvalTime}$$

we can establish and prove an invariant that supports the above safety property, as well as the refinement predicate, and thus prove refinement. This invariant enumerates the reachable control states shown in figure 20. Since there are only 9 such states, the invariant was easily formulated. The proofs for both the invariant and for refinement are again straightforward; the obligations are first split into a separate obligations for each transition, which are then automatically discharged by the decision procedure for linear inequalities with boolean connectives.

In principle, we could have integrated a timed automata tool such as Kronos [53], Uppaal [28], or ATACS [5], into our tool. Although our goal is to make such integration as easy as possible, it would still have required some effort, mainly to define the judgements of these tools in the object logic (roughly *wp* semantics) for ST. On the other hand, the timed system for this proof has a very small number of states, and the required invariant was nearly obvious. This shows one of the strengths of our refinement approach: invariants are often identified and verified at a high level of abstraction where they are free from tedious, "bookkeeping" clutter. The necessary bookkeeping often becomes implicit in the abstraction mappings requiring little or no user effort. Of course, if we found ourselves frequently constructing timing invariants, then the effort to integrate a timed automata tool into our framework could be justified.

6.6 Transistor-Level Implementation

The actual implementation of the divider has been designed in self-timed precharged logic using dual-rail encoded datavalues as described in section 4.3. We used ST to model logic elements implemented in precharged logic; the following example illustrates our modeling approach.

Recall the AND-OR gate from figure 11 in section 4.3. This circuit can be modeled in ST by the five transitions below:

```
   ≪ ¬pb → y.T := FALSE ≫
 ∥ ≪    pb ∧ (a.T ∧ (b.T ∨ c.F))
      → y.T := TRUE
   ≫
 ∥ ≪ ¬pb → y.F := FALSE ≫
 ∥ ≪    pb ∧ (a.F ∨ (b.F ∧ c.T))
      → y.F := TRUE
   ≫
 ∥ ≪ y.empty := NOT (y.T or y.F) ≫
```

This example illustrated the design of a particular logical operation. Other operations are implemented in a similar manner by substituting appropriate pull-down networks for the ones used here. Networks of logical elements are modeled as the asynchronous combination of the corresponding transitions.

A key feature of this approach is that the topologies of the pull-down networks for signals `y.T` and `y.F` are encoded in the guards for the transitions that set these signals to `TRUE`. Because the syntactic structure of the program for the ST model is available in the object logic, it is straightforward to implement inference rules that operate on syntactic encodings such as this encoding of the transistor-level netlist.

To establish that the transistor-level model implements the timed/bit-vector model, two issues 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, and it needs to be shown that the computation performed by the transistor-level model is consistent with the one carried out by the bit-vector models. Second, in the transistor-level model only the quotient digit output is used to determine if a stage has finished evaluation. It therefore needs to be shown that the quotient digit of a stage becomes valid only after all other outputs of a stage are valid. This corresponds to Williams' second optimization as mentioned in section 4.4.

6.6.1 Abstraction Mapping

The first issue is addressed by defining an appropriate abstraction mapping between the dual-rail encoded signals of the implementation and the corresponding binary variables at the bit-vector level. There are two aspects to this abstraction mapping: First, we need to map the four-valued dual-rail codes to binary values. We use the mapping characterized by the following table:

(`y.F`,`y.T`)	$\mathtt{y}'$
(*false*,*false*)	*false*
(*false*,*true*)	*true*
(*true* ,*false*)	*false*
(*true* ,*true*)	*true*

This mapping maps $(\mathtt{y.F}, \mathtt{y.T})$ to `y.F`. Note that the tuple $(true, true)$ is an illegal dual-rail value and as such its mapping is irrelevant[3]. Furthermore, an empty dual-rail value is mapped to *false*. Since the outputs of a bit-vector-model stage are set to all-false-vectors by cell `ClearStage` during its 'precharge' transition (see figure 18), the result of precharging a transistor-level stage will be consistent under this abstraction mapping.

The second aspect of the abstraction mapping relates to the asynchronous update of the individual output signals of a stage at the transistor level: During the evaluation phase, some dual-rail outputs of a stage may already carry valid values, while others are still empty. In the bit-vector implementation however, all the outputs of a stage are updated in one atomic action. Thus, the abstraction mapping must be constructed in a way that ensures that a stage's outputs are only observed when they are meaningful, i.e. when the stage has finished evaluating. This can be accomplished by defining the abstraction mapping such that a stage's outputs are mapped under the dual-rail-to-binary mapping only when the stage satisfies the predicate `holdModeP`, and are mapped to *false* otherwise.

6.6.2 Timing Analysis

The implementation does not include completion detection logic to determine when all of a stage's outputs have assumed a valid dual-rail value. Instead, correct operation rests on the assumption that the output signals for the partial remainder are always set before the quotient digit output. This assumption is based on the observation that the circuitry for the quotient digit (a carry-ripple adder and the quotient-selection logic) is sufficiently many levels 'deeper' than the logic for the remainder (a carry-save adder).

Proving refinement between the timed/bit-vector model and the transistor-level implementation requires showing that this is indeed the case, and furthermore that the implementation satisfies the timing bounds `MinEvalTime` and `MaxPrechargeTime` on evaluation and precharge time stipulated by the timed/bit-vector model.

Using either state-space exploration or deductive reasoning to establish such timing bounds as safety properties of the transistor-level model is impractical. Instead, we make use of the observation that the circuitry of each stage forms an acyclic netlist and is as such subject to well-understood timing analysis techniques.

We implemented a timing analysis procedure that extracts a netlist from an ST program written in the style introduced above, and traverses this netlist to establish bounds on the arrival times of signals at the netlist's outputs, given arrival times at the inputs. The algorithm implements the simple technique of determining the shortest and longest paths in the acyclic signal-dependency graph by a depth first traversal. This is by no means the most sophisticated possible approach possible for timing verification. The simplicity of this algorithm made it easy to implement, unlikely to contain any bugs, and is adequate for verifying the divider. Rather than using a complicated algorithm for timing analysis, we used a simple algorithm augmented by the deductive capabilities of the theorem prover as directed by the user. Details are presented in [34]. The timing analysis procedure is only applicable under certain assumptions, both of static (e.g., the netlist must be acyclic) and dynamic nature (e.g., the netlist's inputs must be stable and valid during evaluation).

[3] It is a corollary of the safety-properties that we establish for the transistor-level model that invalid dual-rail codes can never occur.

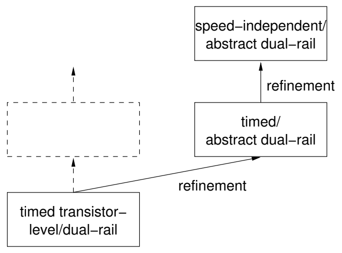

speed−independent/ abstract dual−rail
refinement
timed/ abstract dual−rail
refinement
timed transistor−level/dual−rail

Fig. 21. Side Hierarchy

We encapsulated this timing analysis procedure as a proof rule which introduces the established timing bounds in the form of a safety property. At the same time this rule syntactically verifies the static assumptions and introduces obligations corresponding to the dynamic assumptions, again in the form of safety properties. Furthermore, the rule extracts the functionality of the netlist in the form of Boolean expressions (represented as BDDs) for each output of the netlist as a function of its primary inputs. This provides seamless integration of timing analysis and functional extraction into the deductive refinement argument.

In the verification of the divider, we analyze the transitions for the datapath of a single stage. The user specifies the transitions for one stage, and the netlist is extracted syntactically from the ST source code. The required static conditions are verified automatically from the netlist. Then, lower and upper bounds on the signal arrival times for the stage's outputs are derived, and the required dynamic conditions are introduced as new proof obligations. We describe the verification of these conditions in the next section.

6.6.3 Side Hierarchy

The safety properties established by the above proof rule are sufficient to discharge the refinement predicate between the timed/bit-vector and the transistor-level models. It remains to prove that the assumptions of the timing analysis rule are satisfied. However, the safety properties inherited from the timed/bit-vector model are not strong enough to discharge this obligation, because these properties rest on the correct transitioning between empty and valid dual-rail values, information which is not present at the bit-vector level.

To avoid proving these safety properties directly at the implementation level (which would face similar challenges as proving the refinement predicate directly), we introduced a side hierarchy of models that were designed specifically to allow the inheritance of said safety properties from them.

This side hierarchy, shown in figure 21, consists of two models, referred to as timed/abstract and speed-independent/abstract. Both models have the same timed and speed-independent control transitions as the corresponding timed/bit-vector and speed-independent/bit-vector models. However, instead of the bit-vector data-path, these models contain an abstraction of the transistor-level data-path. Instead of actually computing a remainder and quotient digit, this abstraction consists of only the dual-rail encoded outputs of a stage, which are non-deterministically set to an arbitrary valid value during evaluation. The motivation for this abstraction is straightforward: the actual datapath computes division, an operation that cannot be represented with BDDs and feasible amounts of memory. The non-deterministic abstraction represents a datapath that observes the same self-timed protocols as the actual divider, but for which the actual function computed is unspecified. In effect, the abstract datapath can compute *any* function. The BDD for a completely unspecified function is very compact, allowing efficient verification.

The models in the side-hierarchy contain enough detail to allow the formulation of the desired safety properties, such as 'the outputs of a stage are valid and remain unchanged while its predecessor is evaluating'. Basically, the side-hierarchy mimics the main hierarchy, using dual-rail signals instead of bit-vectors, and leaving the datapath functions unspecified. Thus, the speed-independent, abstract dual-rail model has the same sequencing of precharging, evaluating, and holding operations that we established for the speed-independent model, bit-vector model. In fact, the proofs are very similar and we used much of the original proof for this version. Having established key safety properties for the speed-independent, abstract dual-rail model, we propagate them downward using refinement and use them to establish the required dynamic properties of the timed, transistor-level model. This discharges the side-obligations generated by the timing verification procedure and allows us to establish that the timed, transistor-level model is a valid refinement of the timed, bit-level model.

7 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 are thus established 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.

A key advantage of our refinement based approach is that we can use safety properties that have been proven at higher levels of abstraction when proving correctness at lower levels. As a consequence, we could formulate invariants in models where the needed invariants were fairly simple and obvious, without a myriad of "bookkeeping clauses" that often clutter invariant based arguments. The proof of refinement then becomes one of finding the appropriate abstraction mapping after which the remaining proof obligations often amount to tautologies that can be discharged by decision procedures for boolean formulas and linear inequalities. We note that the bookkeeping clauses are implicitly present in the composition of the abstract invariant with the abstraction function. The user never has to write or see these clauses.

A goal in using domain-specific rules is to make formal verification accessible to a designer who is familiar with traditional timing verifiers, model checkers, etc. To this end, we tried to develop a proof that relies more on designer insight than on subtleties of mathematical logic. Although we believe that we have made significant progress toward this goal, we readily admit that we have not yet achieved it. In particular, a solid understanding of rewriting techniques was needed to formulate the proof that the integer model implements the rational model. Likewise, our design of the side-hierarchy was motivated by an understanding of how bit-vector functions are represented by BDDs. We see our current work as a step towards making formal techniques accessible to a wider range of designers.

Acknowledgments

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. Thanks to Andrew Appel and Lorenz Huelsbergen for answering our questions on integrating the CUDD package into SML. Michael Gordon provided many helpful comments on an earlier version of this paper. Finally, we would like to thank the anonymous referees for their careful and constructive feedback.

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. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
3. 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.
4. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *3rd Int. Symp. on Prog. Lang. Implement. and Logic Program.*, number 528 in Lect. Notes Comput. Sci., pages 1–13. Springer-Verlag, August 1991.
5. W. Belluomini, C. J. Myers, and H. P. Hofstee. Verification of delayed-reset domino circuits using ATACS. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, April 1999.
6. 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. Number 1102 in Lect. Notes Comput. Sci., pages 415–418. Springer-Verlag, August 1996.
7. Manuel Blum and Hal Wasserman. Reflections on the pentium division bug. *IEEE Trans. Comput.*, 45(4):385–393, April 1996.
8. 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.
9. R.S. Boyer and J.S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. Technical Report ICSA-CMP-44, Institute for Computing Science and Computer Applications, The University of Texas, Austin, January 1985.
10. R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, Boston, second edition, 1997.
11. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, August 1986.
12. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, August 1986.
13. J.R. Burch, E.M. Clarke, et al. Symbolic model checking for sequential circuit verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits*, 13(4):401–424, April 1994.
14. Joseph J.F. Cavanagh. *Digital computer arithmetic : design and implementation*. McGraw-Hill, New York, 1984.
15. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994.
16. E.M. Clarke, S.M. German, and X. Zhao. Verifying the SRT divsion algorithm using theorem proving techniques. Number 1102 in Lect. Notes Comput. Sci., pages 111–122. Springer-Verlag, August 1996.
17. E.M. Clarke and X. Zhao. Analytica: a theorem prover for Mathematica. *The Journal of Mathematica*, 3(1), 1993.
18. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
19. 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.
20. 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.
21. Michael J.C. Gordon. Programming combinations of deduction and bdd-based symbolic calculation. In *Proceedings of the Symposium in Celebration of the Work of Tony Hoare*, November 1999.
22. Michael J.C. Gordon. Reachability programming in hol98 using bdds. In *Proceedings of 13th International Conference on Theorem Proving and Higher Order Logics*. Springer, August 2000. LNCS 1869.
23. Cheryl Harkness and Elizabeth Wolf. Verifying the Summit bus converter protocols with symbolic model checking. *Formal Meth. System Design*, 4:83–97, 1994.
24. 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.
25. Institute of Electrical and Electronic Engineers. IEEE standard for binary floating point arithmetic, 1985. Std. 754-1985.
26. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
27. Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.*, 12(3):396–428, July 1990.
28. Kim G. Larsen, Paul Petterson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.
29. Trevor W.S. Lee, Mark R. Greenstreet, and Carl-Johan Seger. Automatic verification of asynchronous circuits. *IEEE Design and Test*, 12(1):24–31, Spring 1994.
30. R.E. Miller. *Switching Theory*. Wiley, New York, 1965.
31. J.S. Moore. Personal communication, 1998.
32. J.S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division program. *IEEE Trans. Comput.*, 47(9):913–926, September 1998.
33. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.
34. 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.
35. S. Owre, J. Rushby, et al. Formal verification for fault tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
36. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Int. Conf. Automated Deduction (CADE '92)*, number 607 in Lect. Notes Comput. Sci., pages 748–752. Springer-Verlag, 1992.
37. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lect. Notes Comput. Sci. Springer-Verlag, Berlin, 1994.

38. Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 2nd edition, 1996.
39. Amir Pnueli and Elad Shahar. A platform for combining deductive with algorithmic verification. Number 1102 in Lect. Notes Comput. Sci., pages 184–195. Springer-Verlag, August 1996.
40. F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In *Proc. EURO-Par '95 Parallel Processing*, number 966 in Lect. Notes Comput. Sci., pages 287–300. Springer-Verlag, August 1995.
41. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In *7th International Conference on Computer Aided Verification*, pages 84–97, 1995.
42. H. Rueß, N. Shankar, and M.K. Shrivas. Modular verification of SRT division. Number 1102 in Lect. Notes Comput. Sci., pages 123–134. Springer-Verlag, August 1996.
43. Charles L. Seitz. System timing. In *Introduction to VLSI Systems* (Carver Mead and Lynn Conway), chapter 7, pages 218–262. Addison Wesley, 1979.
44. Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, April 1979.
45. Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, January 1984.
46. Fabio Somenzi. CUDD: CU Decision Diagram Package. URL: http://bessie.colorado.edu/~fabio/CUDD/cuddIntro.html.
47. 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.
48. Jørgen Staunstrup. *A formal approach to hardware design*. Kluwer Academic Publishers, Boston, 1994.
49. Neil H.E. Weste and K. Eshragian. *Principles of CMOS VLSI Design*. Addison-Wesley, 1993.
50. 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.
51. 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.
52. Stephen Wolfram. *The Mathematica Book*. Cambridge University Press, 1996.
53. Sergio Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2), October 1997.